让开发自动化系列专栏

wizardforcel

Published with GitBook

目錄

让开发自动化系列专栏

> 来源：让开发自动化系列专栏

作为开发人员，我们的工作就是为最终用户实现过程自动化；然而，很多开发人员却忽略了将自己的开发过程自动化的机会。为此，自动化专家 Paul Duvall 编写了 让开发自动化 这个系列文章，专门探讨软件开发过程自动化的实际应用，并教您 何时 以及 如何 成功地应用自动化。

让开发自动化: 部署自动化模式，第 2 部分

更多一键式部署模式

Java™部署常常很混乱，容易出现错误，需要许多手工操作，这会延误向用户交付软件的时间。本文是分两部分的让开发自动化系列文章的第 2 部分。在本文中，自动化专家 Paul Duvall 进一步补充用于开发可靠、可重复且一致的部署流程的一些关键模式，帮助读者为 Java 应用程序生成简便的部署。

关于本系列

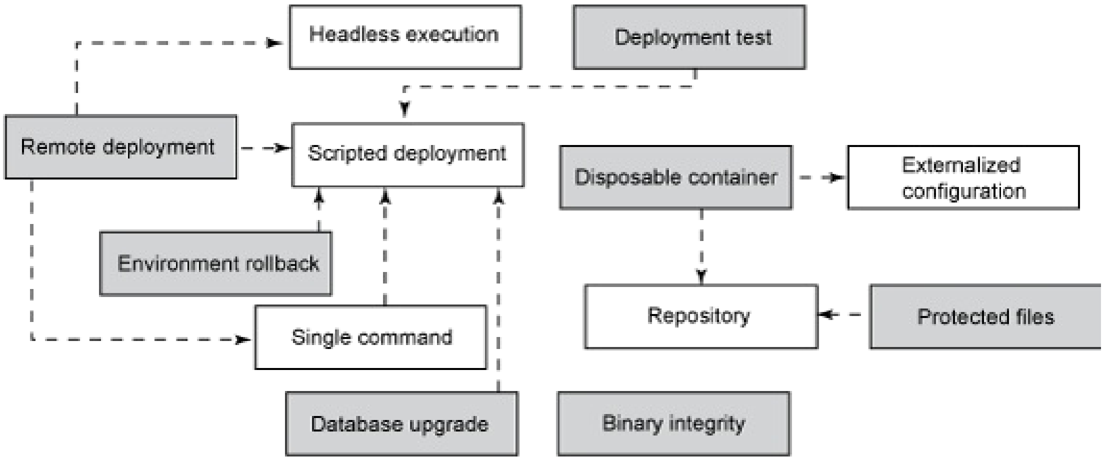
作为开发人员，我们致力于为用户自动化流程；但许多开发人员疏忽了自动化我们自己的开发流程的机会。为此，我们编写了 让开发自动化系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍 何时以及 如何成功应用自动化。

部署是软件创建过程中又一个适合实现自动化的方面。通过自动化部署，可获得一个可靠、可重复的流程，其中好处颇多：更高的准确性、更快的速度和更好的控制。在这个分两部分的系列文章的 第 1 部分中，我描述了 8 种部署自动化模式。在本期，我进一步扩大讨论范围，阐述另外 7 种同样有益的部署方法：

- **Binary Integrity**，确保全部目标环境使用相同的工件。
- **Disposable Container**，使目标环境处于已知状态，以减少部署错误。
- **Remote Deployment**，确保部署可以从一个集中化的机器或集群与多台机器交互。
- **Database Upgrades**，提供一个集中管理的、脚本化流程，以便将增量更改应用到数据库。
- **Deployment Test**，根据最近的部署，使用部署前和部署后检查，确认应用程序按预期运行。
- **Environment Rollback**，如果部署失败，回滚应用程序和数据库更改。
- **Protected Files**，控制对构建系统使用的某些文件的访问。

图 1 解释了本文阐述的部署模式之间的关系（未使用阴影的那些模式在 第 1 部分中介绍过）：

图 **1.** 部署自动化模式



这 7 个附加的部署自动化模式以之前 8 个模式为基础，它们有助于创建一键式（one-click）部署。

一次编译，部署到多个环境

名称：Binary Integrity

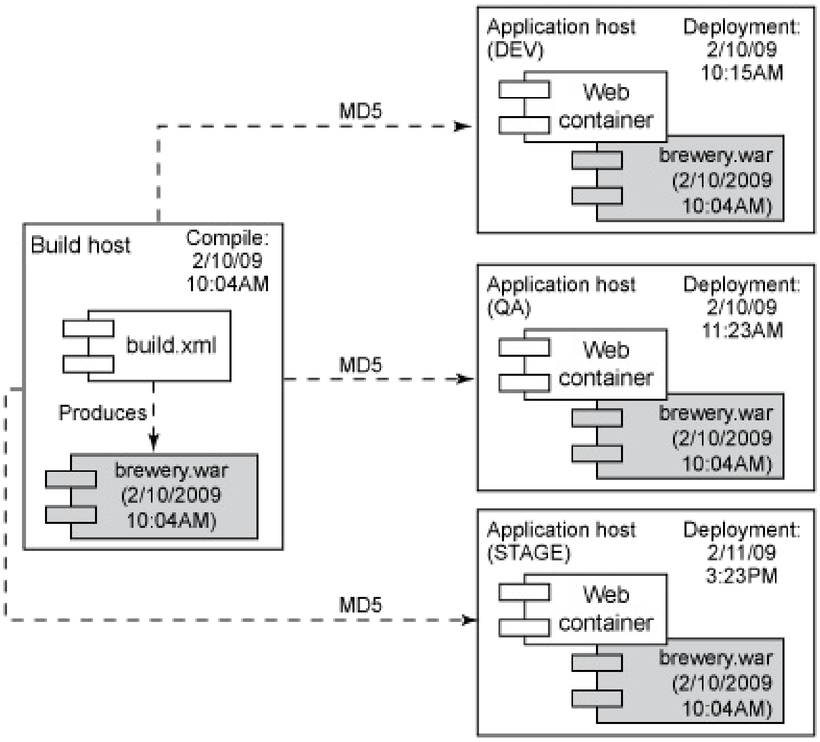
模式：对于每个标记过的部署，每个目标环境中使用相同的归档文件（WAR 或 EAR）。

反模式：对于同一标记，为每个目标环境单独进行编译。

就这个话题与同事经过多次讨论后，我最终站在了 一次编译，部署到多个目标环境一边，而不是 在每个目标环境中编译和打包。例如，一次 Java 开发产生的部署工件是 Web 归档（WAR）或企业归档（EAR）文件。这个归档应该注册到版本控制储存库中，并一次性贴上标记 —就像在 DEV 环境中那样。

图 2 解释了 一次编译，部署到多个环境这一概念：构建机器上生成的同一个 brewery.war 被部署到每个目标环境：

图 **2.** 同一个 **Web** 归档被部署到不同的目标环境



Ant 提供一个 `checksum` 任务 —使用 Message-Digest algorithm 5（MD5）hash 算法 —以确保构建机器上编译和打包的文件就是部署到每个目标环境的文件。

有人会争辩说，虽然工件可能相同，但每个目标环境的部署配置是不同的。也就是说，当使用 Single-Command、Scripted Deployment 时，无论它是否为相同的归档，很多自动化流程可以改变应用程序的输出。确实如此；但是，您还需要花不必要的时间解决问题，因为在 STAGE 环境中使用与 QA 环境不同的 JDK 版本编译和打包软件。并且，当 DEV 中使用的来自一个集中式依赖管理储存库（例如 Ivy 或 Maven）的 JAR 与准备（staging）环境中的那些 JAR 不同时，失败的机率就会增加。这些风险使我确信，为了确保二进制代码的完整性，必须一次性编译和打包，以便部署到多个环境。

用一次性容器降低部署成本

名称：Disposable Container

模式：通过将安装和配置解耦，使 Web 和数据库容器的安装和配置自动化。

反模式：手动将容器安装到每个目标环境并进行配置。

在较早一期的 让开发自动化 文章 “持续集成反模式，第 2 部分” 中，谈到了为什么清理一个 “受污染的” 环境有助于防止出现误判或漏判的构建。Disposable Container 可以减少在使用持久性容器时可能出现的很多问题。Disposable Container 模式基于两个原则：完全移除所有容器组件，以及 将容器的安装与配置分离。对于有些人，尤其是系统工程师来说，这似乎是一

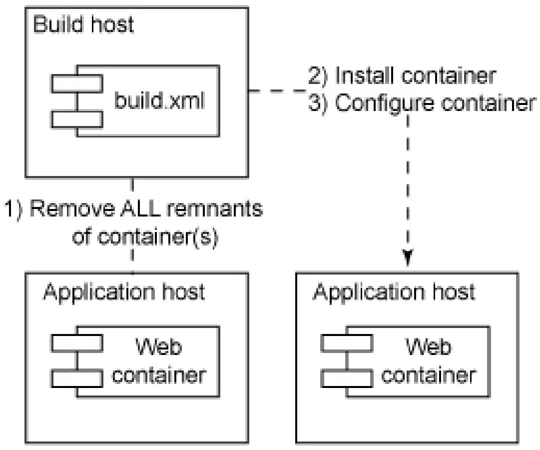
个极端的概念，因为不再要求由一个单独的团队管理和模糊化容器，不让开发人员或其他人接触到它们。然而，考虑到部署期间经常出现的、代价不菲的问题，它可以让所有团队成员的利益最大化。

一键式部署

经常有一些团队和我说："是的，我们已经实现了自动化部署。"当我问一些简单的问题时 — 例如"输入一条简单的命令（例如 `ant`）就可以生成一个有效的软件应用程序吗？"—回答通常是这样的："是的，一旦安装和配置好 Web 容器……"，或者"是的，一旦设置好数据库"。我对于一个真正自动化的部署的定义是，应该能够从一台干净的机器开始，安装 Java 平台和 Ant（有一些方法可以免除这个步骤），然后输入一条 简单命令，即可得到一个可以正常工作的软件应用程序。如果做不到，就不算是"一键式"部署，并且在部署过程中将出现代价不菲的人员方面的瓶颈问题。

如图 3 所示，Disposable Container 模式基于这样一个原则：一切都应该在 系统中 —（使用第 1 部分中提到的 Repository 模式）—而不是在某个人的头脑中。

图 **3.** 部署期间移除和安装容器



清单 1 中的 Ant 脚本从 Internet 下载 Tomcat ZIP，移除之前的部署残留的任何容器，然后解压、安装和启动 Tomcat：

清单 **1.** 用 **Ant** 脚本编写的部署，该脚本移除、重新安装、启动和配置容器

```
<!-- Check to see if Tomcat is running prior to this -->
...
<exec executable="sh" osfamily="unix" dir="${tomcat.home}/bin" spawn="true">
 <env key="NOPAUSE" value="true" />
 <arg line="shutdown.sh" />
</exec>
<delete dir="${tomcat.home}" />
<get src="${tomcat.binary.uri}/${tomcat.binary.file}"
 dest="${download.dir}/${tomcat.binary.file}" usetimestamp="true"/>
<unzip dest="${target.dir}" src="${download.dir}/${tomcat.binary.file}" />
<exec osfamily="unix" executable="chmod" spawn="true">
 <arg value="+x" />
 <arg file="${tomcat.home}/bin/startup.sh" />
 <arg file="${tomcat.home}/bin/shutdown.sh" />
</exec>
<xmltask source="${appserver.server-xml.file}"
 dest="${appserver.server-xml.file}">
 <attr path="/Server/Service[@name='${s.name}']/Connector[${port='${c.port}']"
   attr="proxyPort"
    value="${appserver.external.port}"/>
   <attr path="/Server/Service[${name='${s.name}']/Connector[${port='${c.port}']"
    attr="proxyName"
    value="${appserver.external.host}"/>
</xmltask>
<!-- Perform other container configuration -->
...
<echo message="Starting tomcat instance at ${tomcat.home} with startup.sh" />
<exec executable="sh" osfamily="unix" dir="${tomcat.home}/bin" spawn="true">
 <env key="NOPAUSE" value="true" />
 <arg line="startup.sh" />
</exec>
```

通过使环境处于已知状态，并以一种受控制的方式部署容器，可以减少很多常见的、引发大部分部署难题的部署错误。

在多个外部环境中运行命令

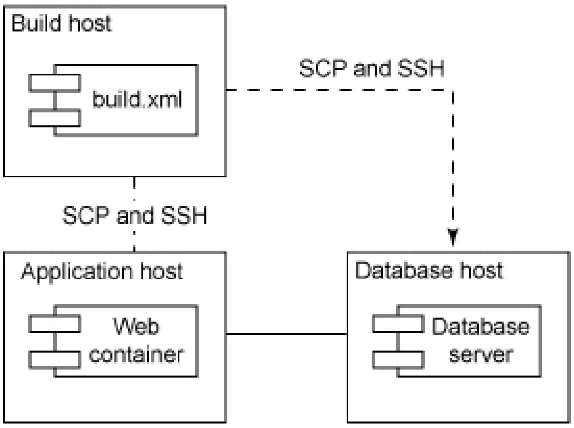
名称：Remote Deployment

模式：使用一个集中式机器或集群将软件部署到多个目标环境。

反模式：在每个目标环境中通过手动方式在本地应用部署。

一旦安装了数据库和 Web 容器，让部署在开发人员的 工作站上运行是件非常简单的事情。然而，开发与生产之间有着巨大的差异。如果组织有多个项目和不同的目标环境（例如测试或准备环境），那么常常需要从一个单独的环境集中地管理部署：一台机器或一个集群。团队常使用一台构建服务器来管理每个目标环境的部署。在 第 1 部分中，我介绍了 Headless Execution 模式，该模式采用公共和私有密钥，所以不必手动登录到每台机器。如图 4 所示，Remote Deployment 依赖于 Headless Execution、Single Command 和 Scripted Deployment 模式，因此可以方便地部署到远程机器：

图 **4.** 用于多个环境的构建管理服务器



要从集中构建服务器远程部署软件，需要使用一些机制安全地进行远程复制和运行命令。我将讨论的两个机制使用 Secure Copy（SCP）和 Secure Shell（SSH）。如清单 2 所示，在 Scripted Deployment 模式下，集中构建机器上生成的 Web 归档被远程地复制到一个目标环境：

清单 **2.** 将 **war** 文件安全地从一台机器复制到另一台机器

```
<target name="copy-tomcat-dist">
 <scp file="${basedir}/target/brewery.war"
 trust="true"
 keyfile="${basedir}/config/id_dsa"
 username="bobama"
 passphrase=""
 todir="pduvall:G0theD!stance@myhostname:/usr/local/jakarta-tomcat-5.5.20/webapps" />
</target>
```

当 WAR 文件被安全地复制到远程目标环境时，我就可以在 Java Secure Channel 中使用 `SSHExec` 之类的任务运行任何 SSH 命令，这一切都是从集中构建机器中远程执行的。另一种方法是 `ssh` 到远程环境，并在本地运行命令。这样可以减少来回的远程传输，并缩短部署时间。

使数据库和数据处于已知状态

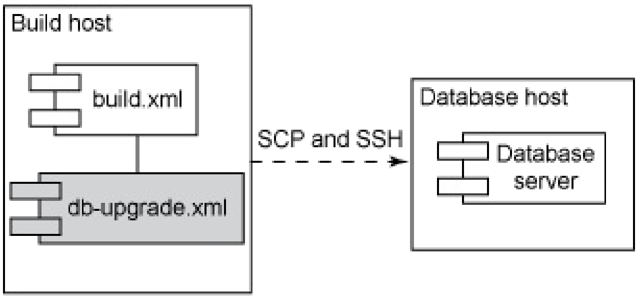
名称：Database Upgrade

模式：使用脚本和数据库在每个目标环境中应用增量更改。

反模式：在每个目标环境中手动应用数据库和数据更改。

在图 5 中，可以看到一个在 Scripted Deployment 中使用自动化脚本更新数据库的例子：

图 **5.** 自动应用增量数据库更新



在较早一期的 让开发自动化文章“实现自动化数据库迁移”中，我谈到了以自动化的方式应用增量数据库更改的必要性。和 Scripted Deployment 中的其他部分一样，数据库更新脚本被签入到储存库中。

LiquiBase（参见 参考资料）是用于将增量更改应用到数据库的工具，可使同样的更改作为 Scripted Deployment 的一部分应用到每一个目标环境。在清单 3 中，一个 SQL 脚本被作为 LiquiBase changelog 的一部分调用。然后，Scripted Deployment（使用一种构建脚本工具实现 —例如 Ant）调用这个 changelog（使用 XML 定义）。

清单 **3.** 从 **LiquiBase** 更改集运行定制的 **SQL** 文件

```
<changeSet id="1" author="jbiden">
 <sqlFile path="insert-distributor-data.sql"/>
</changeSet>
```

要学习和应用自动化数据库更新，还有相当多的事情要做，但其主旨是在 Scripted Deployment 中执行更新，使所有数据库更改都在 系统中，而不是一个写好的程序或存在某个人的头脑中。

冒烟测试部署

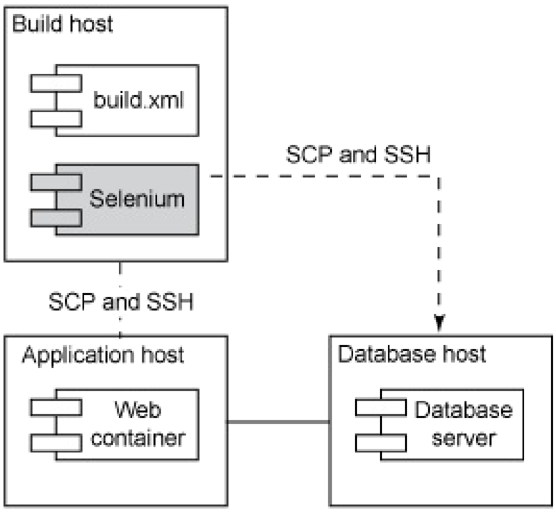
名称：Deployment Test

模式：将自测试功能编写到 Scripted Deployments 中。

反模式：通过运行手动功能测试来验证部署，没有关注特定于部署的方面。

图 6 解释了在部署前后运行部署测试的一个例子：

图 **6.** 对应用程序运行功能部署测试



在清单 4 中，我使用 Ant 执行部署前测试，以确认正在使用的版本是正确的工具版本。在 Scripted Deployment 中，脚本可以检查正在使用的端口（这可能导致 Web 容器部署失败），检查与数据库的连接，检查容器是否已被启动，以及很多其他内部部署测试。

清单 **4.** 运行部署前检查，确保部署有效

```
<condition property="ant.version.success">
 <antversion atleast="${ant.check.version}" />
</condition>
<antunit:assertPropertyEquals name="ant.version.success" value="true" />
<echo message="Ant version is correct." />
<echo message="Validating Java version..."/>
<condition property="java.major.version.correct">
 <equals arg1="${ant.java.version}" arg2="${java.check.version.major}" />
</condition>
<antunit:assertTrue message="Your Java SDK version must be 1.5+. \
 You must install correct version.">
 <isset property="java.major.version.correct"/>
</antunit:assertTrue>
```

更全面的部署测试可以确保应用程序的 功能性是正确的。通过使用用于 Web 应用程序的 Selenium 或用于客户机应用程序的 Abbot 之类的工具编写 特定于部署的自动化功能测试，可以验证是否已正确应用了部署更改。可以将这些测试看作 冒烟测试（*smoke tests*）：只需测试受部署影响的功能。例如，表 1 展示了使用 Selenium 和其他用于 Web 应用程序的工具的一些方式：

表 **1.** 部署测试

部署测试	描述
数据库	编写一个自动化功能测试，该测试将数据插入到数据库。验证数据是否被输入到数据库中。
简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）	编写一个自动化功能测试，该测试从应用程序发送一个电子邮件消息。
Web 服务	使用 SoapAPI 之类的工具提交一个 Web 服务，并验证输出。
Web 容器	验证所有容器服务是否正确运行。
轻量级目录访问协议（Lightweight Directory Access Protocol，LDAP）	使用应用程序，通过 LDAP 进行验证。
日志记录	编写一个测试，该测试使用应用程序的日志记录机制编写日志。

自动化测试不仅仅用于测试用户功能。通过创建侧重于部署测试的套件，可以检验部署的有效性，减少下游错误和开发成本。

回滚所有部署更改

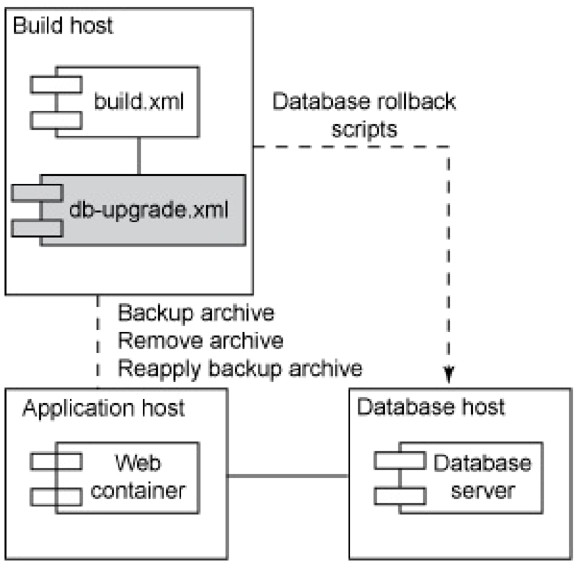
名称：Environment Rollback

模式：当部署失败后，提供自动的 Single Command 更改回滚。

反模式：手动回滚应用程序和数据库更改。

图 7 解释了回滚数据库更改 —使用 Database Upgrade —以及回滚 Web 部署的自动化过程：

图 **7.** 回滚部署更改



不管是否执行自动化部署，当部署失败时，最好有一种方式可以回滚更改。在某些情况下，错误的更改可能导致系统中断，使组织损失数百万美元。要执行 Environment Rollback，需要让目标环境回到部署前的状态。为此，实际上每个更改都需要一个回滚脚本。Web 部署常常需要回滚更多的更改。Environment Rollback 的一个例子是在部署前复制归档（例如一个 WAR 文件），并为每个更改提供回滚数据库脚本。另外还需要重新应用已应用于 Web 容器的配置更改。

清单 6 演示了一个示例，该示例使用 LiquiBase 为每个前滚语句提供一个回滚语句。我将添加一个名为 `brewery` 的新表，同时提供一个相应的 `dropTable` 回滚语句。

清单 **6.** 当应用增量数据更新时提供回滚过程

```
<changeSet id="rollback-database-changes" author="bobama">
 <createTable tableName="brewery">
   <column name="id" type="int"/>
 </createTable>
 <rollback>
   <dropTable tableName="brewery"/>
 </rollback>
</changeSet>
```

这个简单的例子仅用于说明问题，并不意味着回滚就是这么简单。恢复到前一个部署常常是一个复杂的、费时的过程（需要实现自动化）。用于编写回滚脚本的时间应该与部署失败付出的代价成比例。

保护信息不被窥探

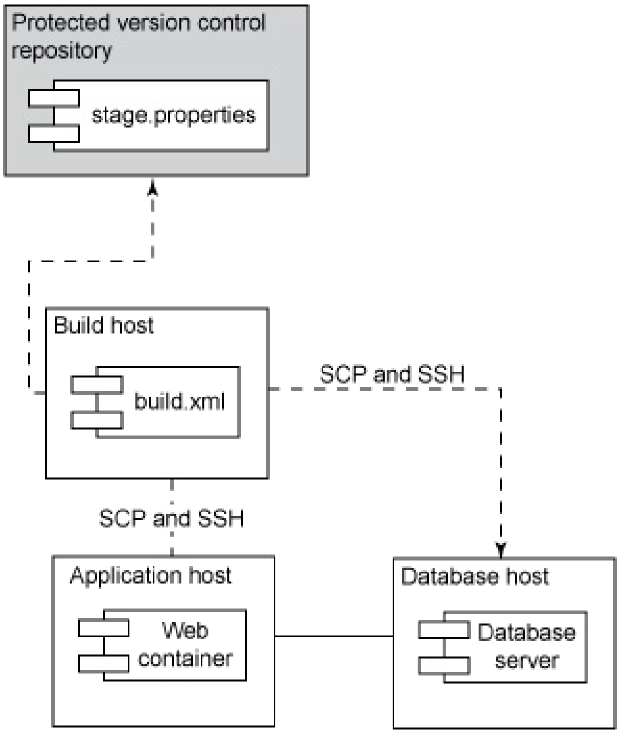
名称：Protected Files

模式：使用储存库，只允许经过授权的团队成员共享文件。

Antipattern：在团队成员的机器上管理文件，或者将文件存储在可由已授权团队成员访问的共享驱动器上。

图 8 展示了一个受保护的版本控制储存库，它用于存放只有已授权人员或系统可以访问的文件：

图 **8.** 使用受保护的版本控制储存库存放敏感文件



在某些情况下，并不是所有团队成员都应该访问特定于环境的数据。但是，将该信息与部署脚本分离又可能使脚本无法执行。当讨论 Headless Execution 模式时，我描述了使用 SSH 密钥和 Java Secure Channel 工具复制文件，并安全全地运行远程命令，而不需要人为输入命令。使用 Externalized Configuration 处理的属性很可能包含不应该让所有团队成员看到的数据。为了确保实现 Headless Execution，同时防止 .properties 文件中的数据被窥探，我使用的技巧是将这些文件签入到一个受保护的储存库中。

在清单 7 中，我配置了一个由 Apache 托管的 Subversion 储存库，先拒绝所有用户访问某个目录，然后显式地添加某些用户：

清单 **7.** 在 **Apache** 上使用 **Subversion** 保护一个 **Subversion** 储存库

```
<DirectoryMatch "^/.*/(\.svn)/">
 Order deny,allow
 Deny from all
 Allow bobama,jbiden,hclinton
</DirectoryMatch>
```

通过保护对 Subversion 储存库的访问，可以将一个 Scripted Deployment 设为被允许的用户，使其不必输入密码就可以访问属性，从而通过 SSH 密钥定义的方式实现 Headless Execution。

一键式部署

除了这份分两部分的系列文章中描述的 15 个部署自动化模式外，我还归纳了更多的模式，但是这 15 个模式大概可以应对我遇到的 80% 的部署情况。每个模式都是为了帮助在每个目标环境中实现真正的一键式 / 单命令部署。希望您一切顺利！

结束语

这是我的 让开发自动化系列的最后一篇文章。在两年多的时间里与您一起分享我的经验，令我感觉像在经历一场有趣的探险。编写这个系列的目的是展示如何以及为何自动化大量软件开发过程，使开发人员可以将更多的时间花在感兴趣的问题上，而不是将时间浪费在重复的、容易出错的活动上。在这个系列中，我演示了如何自动化代码检查以便适当地重构、增量式地升级应用程序数据库、应用 Continuous Integration 实践和工具、每次更改时运行自动测试、生成 GUI 安装程序、创建一键式部署、使开发人员自动生成文档、执行依赖管理、利用版本控制储存库，以及有效地使用各种构建脚本和工具。希望您在阅读这个系列时能够充满乐趣。

让开发自动化: 部署自动化模式，第 1 部分

用于实现简便部署的模式

Java™ 部署常常很混乱，容易出现错误，需要许多手工操作，这会延误向用户交付软件的时间。本文是分两部分的 让开发自动化 系列文章的第 1 部分。在本文中，自动化专家 Paul Duvall 将介绍用于开发可靠、可重复且一致的部署流程的一些关键模式，帮助读者为 Java 应用程序生成简便的部署。

软件部署常常被视为不可避免的麻烦，可以在遇到它时应付一下，以后就不用理会了。但是，与开发周期的其他部分一样，可以并且应该对部署应用软件工程原理。在手工进行部署时，部署是一个重复且容易出现错误的流程。正如可以通过自动化构建来减少错误并加快软件开发，也可以通过自动化部署流程来减少错误和加快软件交付。

在前面的一期让开发自动化 “使用自动化加速部署” 中，介绍了一种把软件远程部署到多个目标环境中的技术。本文在更高的层面上讨论自动化部署。正如存在一些用于软件开发的模式，也有一些用于部署的模式。在过去几年里，我一直在收集和整理这些模式。在第 1 部分中，我将介绍 8 个部署模式并提供相关示例：

- **Repository**，这个模式在一个集中的存储库中管理所有配置文件，这样就可以使用 Scripted Deployment 生成有效的软件
- **Scripted Deployment**，这个模式通过脚本执行所有部署操作，这样在执行部署时就不需要人为干预
- **Single Command**，这个模式减少部署的复杂性，确保实现部署流程的无头执行（Headless Execution）
- **Tokenize Configuration**，这个模式提供一种可重复的把可变信息注入配置文件中的方法
- **Externalized Configuration**，这个模式可以一次性地 输入在目标环境之间有差异的信息
- **Template Verifier**，这个模式帮助确保所有目标环境属性都是相同的
- **Headless Execution**，这个模式提供一种在自动化流程中安全地访问多台机器的方法
- **Unified Deployment**，这个模式帮助建立可以在许多目标环境中运行的单一部署脚本

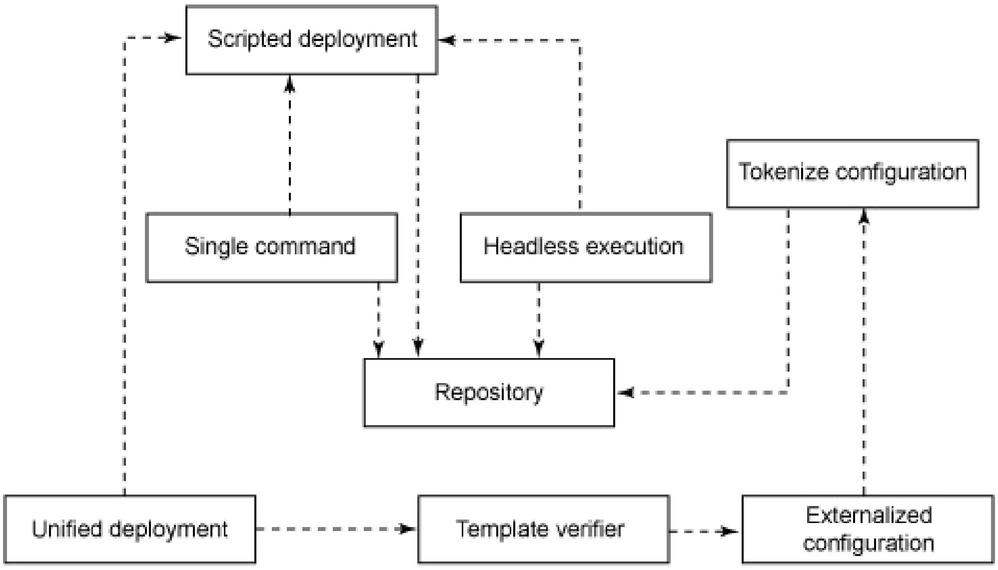
第 2 部分将介绍更多的部署模式。

关于本系列

作为开发人员，我们致力于为用户自动化流程；但许多开发人员疏忽了自动化我们自己的开发流程的机会。为此，我们编写了 让开发自动化 系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍何时 以及如何 成功应用自动化。

图 1 说明本文讨论的部署模式之间的关系：

图 **1.** 部署自动化模式



我将依次讨论每个模式。在这个过程中，您会逐渐理解图 1 中显示的关系。

把所有文件提交给一个版本控制存储库

名称：Repository

模式：所有文件都被提交给版本控制存储库 — 在部署上下文中，这是指所有配置文件 和工具。

反模式：一些团队把这些信息放在一个具有访问控制的共享驱动器上。其他团队可能仅仅把信息放在自己的机器上，然后把它们复制到目标环境中。

作为一般规则，我建议开发团队签入创建软件所需的所有 文件。有时候，这个规则有一些例外情况，但是不常见。在部署上下文中，一些团队错误地认为服务器和服务器配置是固定不变的资产。尽管有时候签入大的二进制代码比较困难，但是应该把配置、数据库脚本以及所有构建和部署脚本提交给版本控制存储库。使用 Repository 模式对于实现下面讨论的 Scripted Deployment 和 Single Command 模式很有帮助。

用脚本执行所有部署流程

名称：Scripted Deployment

模式：在脚本中编写所有部署流程。

反模式：一些团队可能手工配置部署任务，比如安装和配置 Web 容器。其他团队可能使用容器提供的基于 GUI 的管理工具针对特定环境修改容器。尽管这种方式最初可以简化配置的手工修改，但是如果要在许多目标环境中频繁地执行部署，这就不合适了。另外，在首次 使用时，基于 GUI 的管理工具对于部署非常有帮助。但是，因为许多人必须重复执行这些流程，所以这种方式的可伸缩性很差且容易出现错误。

清单 1 通过实现 Scripted Deployment 模式对启动（或重新启动）Tomcat Web 容器的流程进行自动化。这个流程是用 Apache Ant 构建脚本语言编写的。

清单 **1.** 启动 **Tomcat Web** 容器的示例

```
<available file="@{tomcat.home}/server/@{tomcat.server.name}/bin"
   property="tomcat.bin.exists"/>
<if>
  <isset property="tomcat.bin.exists"/>
<then>
  <echo message="Starting tomcat instance at @{tomcat.home} with start_tomcat" />
  <exec executable="@{tomcat.home}/server/@{tomcat.server.name}/bin/start_tomcat"
   osfamily="unix" />
</then>
<else>
  <echo message="Starting tomcat instance at @{tomcat.home} with startup.sh" />
  <exec osfamily="unix" executable="chmod" spawn="true">
    <arg value="+x" />
    <arg file="@{tomcat.home}/bin/startup.sh" />
    <arg file="@{tomcat.home}/bin/shutdown.sh" />
  </exec>

  <exec executable="sh" osfamily="unix" dir="@{tomcat.home}/bin" spawn="true">
    <env key="NOPAUSE" value="true" />
    <arg line="startup.sh" />
  </exec>

    <exec osfamily="windows" executable="cmd" dir="@{tomcat.home}/bin" spawn="true" >
      <env key="NOPAUSE" value="true" />
        <arg line="/c startup.bat" />
    </exec>
    <sleep seconds="15" />
    </else>
  </if>
```

通过脚本执行这个流程，就不再需要在 Tomcat 提供的 GUI 管理界面中执行操作。另外，因为这个流程已经脚本化了，可以在全面的自动化部署中通过无头流程运行它。

通过单一命令运行部署

名称：Single Command

模式：部署者或无头流程只需输入单一命令，即可为用户生成有效的软件。

反模式：某些部署流程要求部署者输入多个命令并完成多个流程，比如复制文件、修改配置文件、重新启动服务器、设置密码等等，这些重复的操作很容易出现错误。如果他们走运，手头有详细的文档，能够指导他们执行这些操作。但是无论如何，要求人员执行部署流程都

会增加出错的风险，导致时间瓶颈，延误软件在多个目标环境中的发布。

开发环境不等于生产环境

即使软件能够在开发环境或 QA 环境中正常工作，也不意味着它会在生产环境中正常工作，因为环境之间可能存在各种差异。这就是为什么通过脚本执行所有部署工作是很重要的。

在编写部署时，您的客户常常是团队、组织和公司中的其他人 — 甚至可能是计算机。运行部署的方法越复杂，别人或无头流程成功执行它的可能性就越小。清单 2 给出一个简单的单一命令部署示例：

清单 **2.** 使用 **Ant** 执行单一命令部署

```
ant -Dproperties.file=$USERHOME/projects/petstore/properties/dev-install.properties \
  deploy:remote:install
```

清单 2 中的命令执行一个名为 `deploy:remote:install` 的 Ant 任务，并传递一个与环境相关的 .properties 文件，从而把软件远程部署到其他计算机上。这个任务执行的操作包括使用 Secure Copy protocol (SCP) 安全地复制文件；通过 Secure Shell (SSH) 在远程计算机上安全地执行命令；安装、配置和重新启动 Web 容器；以及其他流程 — 这些都不需要人为干预。显然，用户可以输入这个命令；但是因为它非常简单，所以很容易通过无头流程（比如持续集成或构建管理服务器）运行它。

把可变信息注入配置文件

名称：Tokenize Configuration

模式：把标记化的值输入配置文件，然后在 Scripted Deployment 期间根据 Repository 中的 Externalized Configuration 属性替换它们。

反模式：把与目标相关的数据输入每个环境中的配置文件。

清单 3 中的 XML 文件用来管理 Web 容器和数据库服务器之间的配置。在这个文件中，我使用 `@` 符号设置了一些标记。在自动化部署期间，一个脚本将把这些标记替换为来自 Externalized Configuration 文件的实际值。

清单 **3.** 标记化的 **Web** 容器配置文件

```
<datasources>
  <local-tx-datasource>
    <jndi-name>@application.context.name@</jndi-name>
    <use-java-context>false</use-java-context>
    <connection-url>@database.url@</connection-url>
    <user-name>@database.user@</user-name>
    <password>@database.password@</password>
    <driver-class>@database.driver@</driver-class>
  </local-tx-datasource>
</datasources>
```

把与环境相关的值标记化，就使 Scripted Deployment 能够通过 Unified Deployment 支持多个环境。

提取所有与环境相关的属性

名称：Externalize Configuration

模式：把所有可变值从应用程序配置转移到构建时属性中。

反模式：一些团队为每个目标环境手工硬编码这些值，或者使用 GUI 工具执行同样的工作。

清单 4 给出一些常常包含在与应用程序相关的配置文件中的属性。通过把所有可变值 集中在一个 .properties 文件，就可以把数据（可变属性）与行为（部署脚本）分隔开。换句话说，无论在哪个目标环境中，自动化部署流程都以相同的方式运行。

清单 **4.** 可以提取到与应用程序相关的文件中的示例属性

```
authentication.type=db
application.url=http://${tomcat.server.hostname}:${tomcat.server.port}/brewery-webapp
database.type=mysql
database.server=localhost
database.port=3306
database.name=mydb
database.user=myuser!
database.password=mypa$$!
database.url=jdbc:mysql://${database.server}:${database.port}/${database.name}
tomcat.server.hostname=localhost
tomcat.server.name=default
tomcat.web.password=pa$$123!
tomcat.cobraorb.port=12748
```

清单 4 中的值常常分散在源代码、服务器配置、XML、.properties 和其他文件中。另外，我发现这些数据常常在系统中重复出现，这会导致难以调试的部署问题。通过把这些信息从许多地方集中到一个 .properties 文件中，可以减少发生部署问题的可能性。

通过无头执行减少麻烦

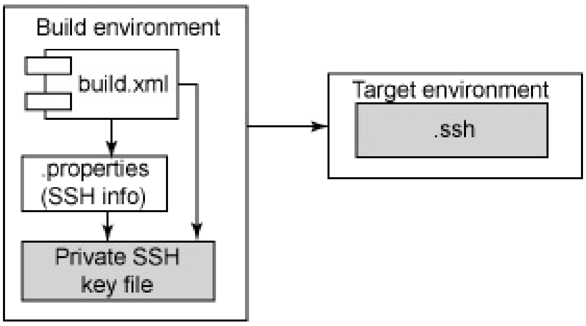
名称：Headless Execution

模式：安全地与多台计算机进行交互，而不需要输入任何命令。

反模式：由人以不同的用户身份手工登录并访问每台计算机，然后执行复制文件、配置值等操作。

尽管 “无头执行” 这个名称有点儿夸张，但是在需要通过自动化流程远程访问其他计算机时，无头执行确实是很有效的解决方案。通过使用公共密钥基础结构 (PKI)，可以把通常由开发人员、构建工程师、软件配置管理 (SCM) 或操作员执行的命令组合为自动化的解决方案。在图 2 中，在构建计算机和 SSH 上安装一个私有密钥文件。在每个与目标相关的 .properties 文件中定义具体值。这通常包括私有密钥文件名和位置、SSH 端口号和主机名。目标计算机包含公共 SSH 密钥文件，从而完成 SSH 握手。

图 **2.** 使用 **SSH** 密钥实现 **Headless Execution** 模式



如果使用这种方式，Scripted Deployment 就可以从构建环境对目标环境执行部署流程，不需要人为干预。

检查在不同的环境中属性是否相同

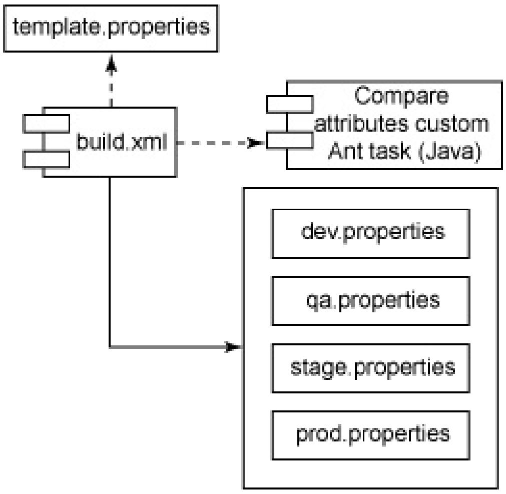
名称：Template Verifier

模式：创建一个模板文件，所有目标环境属性都基于这个文件。

反模式：一些团队执行手工检查、尝试和纠错（在部署失败时，检查失败的原因），或者把文件 “隐藏” 在计算机上。

问题在于，需要确保在每个环境中具有完全相同的属性。但是，在自动化环境中如何检查这一点呢？根据一个模板 .properties 文件检查所有目标环境文件，就可以确保所有属性在任何目标环境中都是相同的。在图 3 中，构建脚本运行一个 Ant 任务，此任务对 template.properties 和与目标相关的 .properties 文件（dev.properties、qa.properties 等）中的属性（而不是值）进行比较。如果发现差异，部署就会失败。

图 **3. Template Verifier** 模式的实现



清单 5 给出一个示例 template.properties 文件。注意，它只包含属性，因为值与这个问题无关。

清单 **5.** 包含属性但不包含值的模板文件

```
db.database=
db.username=
db.password=
db.hostname=
db.driver=
db.port=
db.url=
```

清单 6 给出 dev.properties（或 qa.properties 等）文件的片段。注意，它既包含属性，也 包含值。这些值是与目标环境相关的。

清单 **6.** 基于模板文件的目标环境属性文件

```
db.database=brewery
db.username=root
db.password=p@ssword
db.hostname=dev1.domain.com
db.driver=com.mysql.jdbc.Driver
db.port=3306
db.url=jdbc:mysql://${db.hostname}:${db.port}/${db.database}
```

同时在多个目标环境中执行部署

名称：Unified Deployment

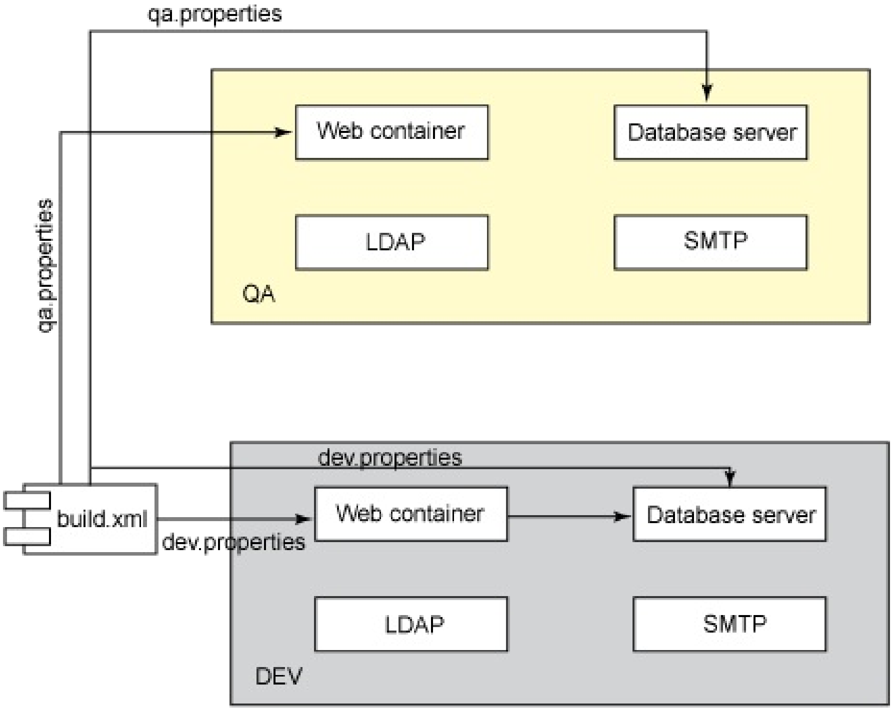
模式：创建能够在不同的平台和目标环境中运行的单一部署脚本。

反模式：一些团队为每个目标环境（甚至特定的计算机）使用不同的部署脚本。

尽管一些部署流程只在某些环境中运行，但是所有流程应该能够 在任何 目标环境中运行。例如，在开发、测试、准备和生产环境中应该运行相同的 Scripted Deployment，但是使用不同的 Externalized Configuration 文件。使用 Template Verifier 检查 Externalized Configuration 属性。

图 4 给出一个能够在多个目标环境中执行部署的部署脚本：

图 **4.** 单一部署，多个目标环境



结束语

部署是软件开发中另一个适合自动化的方面。自动化部署可以受益于可靠且可重复的流程：提高准确性、速度和控制能力。在本文中，我讨论了 8 个适用于软件部署自动化的模式。第 2 部分将讨论更多的模式，包括 Remote Deployment、Disposable Containers、Deployment Test 和 Environment Rollback。

让开发自动化: 使用基于向导的安装程序

使用 *IzPack* 生成 *GUI* 安装程序

对于大多数用户来说，安装软件常常是一件痛苦的事情。生成安装包是软件开发的 “最后一步”，但它可能导致不同的结果：要么用户采用软件，要么它就成为无人问津的垃圾品。在本期的 让开发自动化中，自动化专家 Paul Duvall 演示了如何使用免费、开源的工具 IzPack 来编写为您的用户安装软件的基于向导的安装程序。

在我职业生涯的大部分时间里，我参与了软件开发的整个生命周期 —不仅设计软件的需求、设计、开发和测试，还涉及部署、构建管理、文档编制和安装等活动。最近，随着敏捷开发越来越流行，这些活动可能会更加规范。然而，我在敏捷项目方面的经验表明，有效的部署和安装并没有受到同等的重视。这很滑稽，因为如果潜在用户不能轻松地安装您的软件，那么您很可能会失去他们。提供一种简单的方式来安装您的软件，这对于吸引和留住用户至关重要。

关于本系列

作为开发人员，我们致力于为用户自动化流程；但许多开发人员疏忽了自动化我们自己的开发流程的机会。为此，我们编写了 让开发自动化系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍 何时以及 如何成功应用自动化。

这些年来，我使用过很多安装程序工具。我们团队在今年年初开始的一个大型项目中，为了创建企业级安装程序，必须满足一些非常特别的需求。我们看过 Antigen、AntInstaller、Denova、install4j、InstallAnywhere、IzPack、NSIS 等工具。但是根据项目的特定需求，我们最后决定使用 IzPack，因为：

- 它可以在多种平台上运行。我们需要支持 Windows®、Linux®和 Macintosh。
- IzPack 使用 Java™语言，而我们团队对于这种语言有丰富的经验。
- 它可以执行 Apache Ant 脚本。我们已经花了大量的时间为软件部署而编写 Ant 脚本。
- IzPack 可免费下载。

IzPack 在 2001 年已经出现。它为创建基于向导的安装程序提供了一套丰富的特性。在本文中，我演示如何使用该工具创建安装程序，并给出定义面板、用脚本编写验证器、设置资源等方面的例子。

下载和安装 IzPack

IzPack 文档

要仔细查看的第一个子文件夹是 doc 文件夹。它包含 HTML、PDF 和 Javadoc 版本的文档，当您查阅如何在 IzPack 中编写脚本时，它们是您忠实的伙伴。

下载和安装 IzPack 非常简单。IzPack 使用 IzPack 来安装 IzPack，这也许并不奇怪。请访问 IzPack 网站，下载 IzPack JAR 文件（请参阅 参考资料）。

要安装 IzPack，必须有一个正在运行的 Java Runtime Environment（JRE）。打开命令行提示符，输入 `java -jar IzPack-install-4.1.0.jar` ，可以根据需要修改版本。

基于向导的安装程序要求提供基本的安装信息，比如想将 IzPack 安装在哪里。IzPack 安装完毕后，将得到一个可以运行的示例安装程序。

修改示例脚本

IzPack 提供了一套完整的示例安装脚本。以它们为基础编写自己的安装程序，这是获得一个可运行的安装程序的最快方法。安装 IzPack 的根目录下有一些子目录，包括 bin、doc 和 lib 等。示例安装程序在子文件夹 sample 中，实际上它包含了开发您自己的安装程序所需的所有东西。我选择复制这个 sample 目录，这样就可以随意修改它，而不会破坏原始的内容。图 1 显示了 sample 目录的内容：

图 **1. IzPack** 文件的文件列表

doc		File Folder
listener		File Folder
src		File Folder
antActionSpec.xml	3 KB	EditPlus XML (.xml)
install.jar	609 KB	Executable Jar File
install.xml	4 KB	EditPlus XML (.xml)
Licence.txt	1 KB	Text Document
Readme.txt	1 KB	Text Document
script.bat	1 KB	MS-DOS Batch File
userInputSpec.xml	29 KB	EditPlus XML (.xml)

下面是对图 1 中列出的每个文件的描述：

- **antActionSpec.xml**：执行与构建相关的 Ant 脚本的文件。
- **install.jar**：IzPack 编译期间生成的 JAR 安装文件。这是用户运行的安装程序文件。
- **install.xml**：IzPack 的主安装脚本。IzPack 安装程序中使用的所有资源都以这个脚本开始。
- **Licence.txt**：安装程序的许可文件。
- **Readme.txt**：软件用户使用的 readme 文件。
- **userInputSpec.xml**：IzPack 特有的 XML 脚本，用于定义用户在一个安装程序面板中输入信息时的行为（验证、默认值、字段大小等）。

接下来，通过查看 install.xml 脚本仔细研究 IzPack。

资源

通过资源定义不同的脚本、图像、许可和其他文件，它们共同构成我将要创建的安装程序。我在 install.xml 脚本中定义了一个 `<resources>` XML 元素。在 `<resources>` 元素下，我可以定义安装程序将使用的多个文件，如清单 1 所示：

清单 **1.** 在 **IzPack install.xml** 中定义资源文件

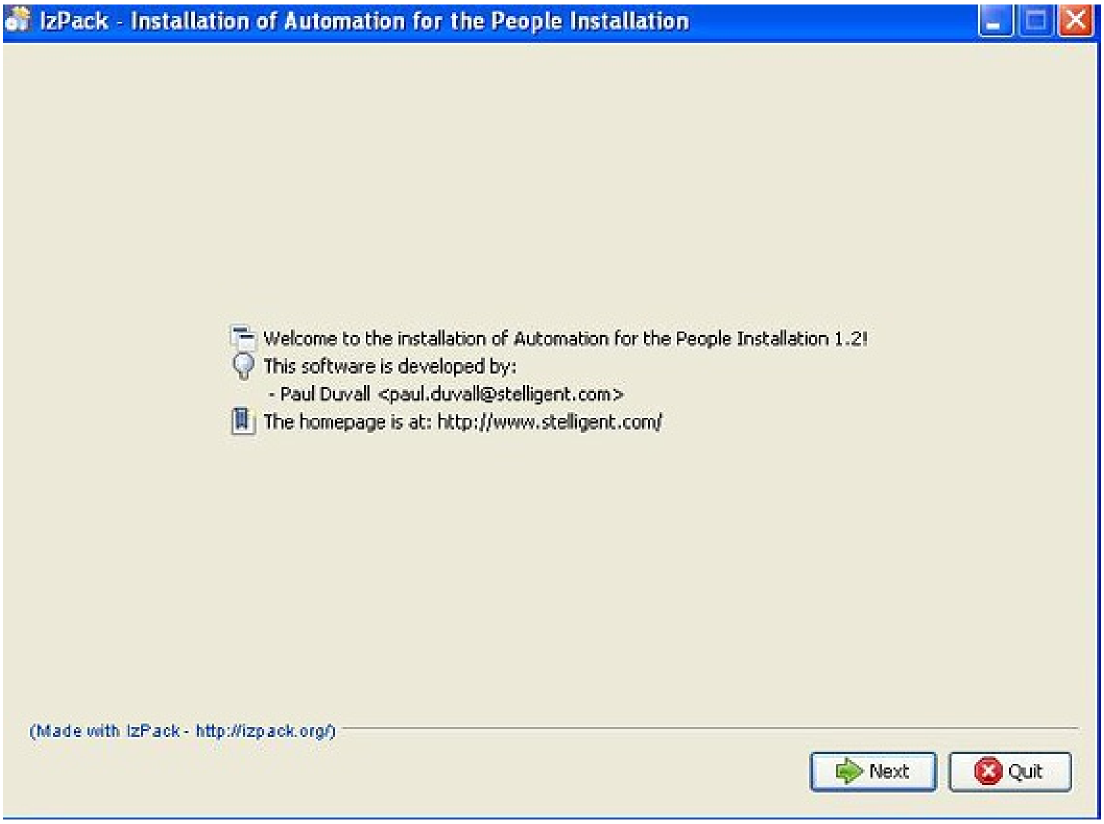
```
<resources>
  <res id="LicencePanel.licence" src="Licence.txt"/>
  <res id="InfoPanel.info" src="Readme.txt"/>
  <res id="AntActionsSpec.xml" src="antActionSpec.xml" />
  <res id="userInputSpec.xml" src="userInputSpec.xml" />
 </resources>
```

可以将 IzPack 的资源看作安装程序的“原料单”，其中定义了用于安装程序的所有文件。

面板

面板是用户在安装向导的每一步中看到的东西。IzPack 提供了很多类型的开箱即用的面板，您可以根据自己的需求定制它们。在图 2 中，我定制了 IzPack 的 `HelloPanel`，以便向用户提供介绍信息：

图 **2. IzPack** 提供的定制的基于向导的 **GUI** 安装



标准面板包括 LicensePanel 、 UserInputPanel 和 PacksPanel 等。在 install.xml 文件中，可以使用 <panels> 元素定义要显示的面板，然后再定义编写安装程序时将使用的面板模板。清单 2 中的例子演示了如何定义面板模板：

清单 **2.** 确定安装程序中显示的面板

```
<panels>
 <panel classname="HelloPanel"/>
 <panel classname="InfoPanel"/>
 <panel classname="LicensePanel"/>
 <panel classname="UserInputPanel" id="UserInputPanel.0" />
 <panel classname="TargetPanel"/>
 <panel classname="PacksPanel"/>
 <panel classname="InstallPanel"/>
 <panel classname="FinishPanel"/>
</panels>
```

您最常用的是模板类型可能是 UserInputPanel 。这是让用户可以输入可变信息而定制的面板模板。这包括用户的联系方式信息、认证凭证、目录位置等。用户可能需要根据他们特定的环境在面板上输入信息。由于我们的团队需要用户连接到一个数据库并设置多个 JBoss 容器，所以我们使用面板提示用户提供特定的信息。

清单 3 是从 userInputSpec.xml 摘录的一个例子，在 清单 1中我将它定义为一个资源。在这个例子中，我将收集特定于用户的信息，用于连接到一个特定的数据库。

清单 **3.** 定义安装程序中的面板的属性

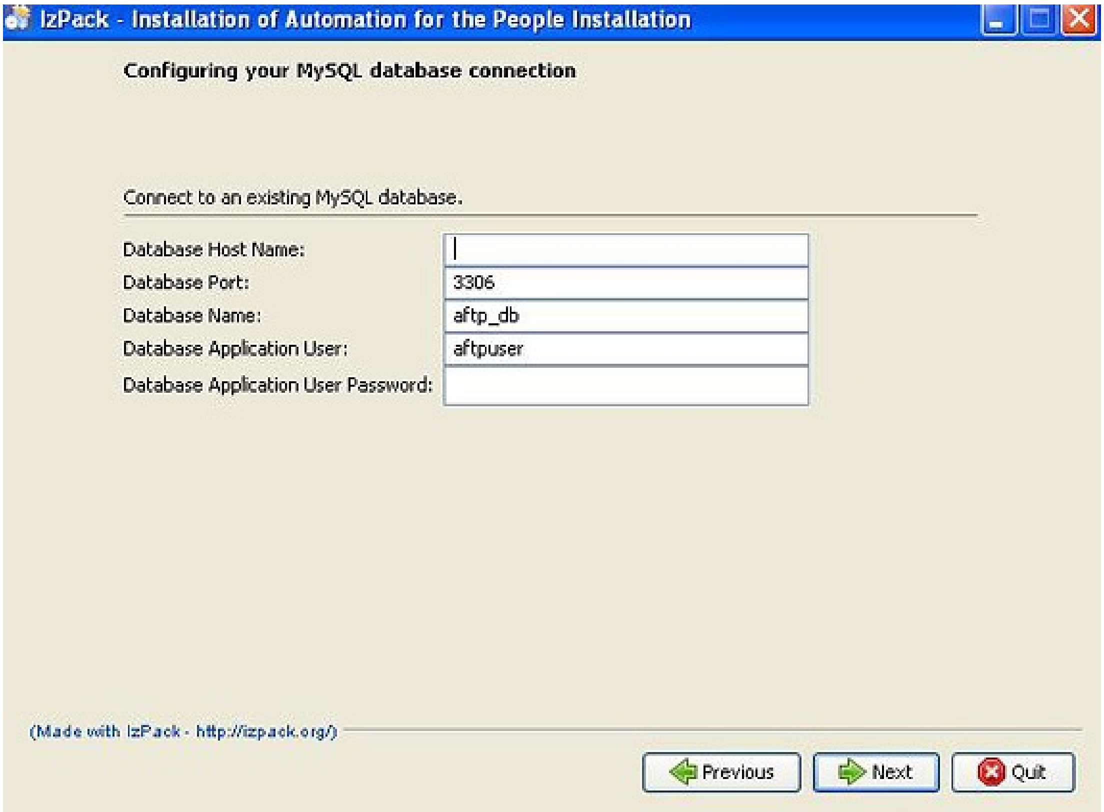
```
<panel order="0">
 <field type="title" txt="Configuring your database connection" bold="true" size="1" />
 <field type="staticText" align="left" txt="Connect to an existing database..."/>
 <field type="divider" align="top"/>

 <field type="text" variable="database.hostname">
  <spec txt="Database Host Name:" id="databasehostname.label" size="40" set=""/>
  <validator class="com.izforge.izpack.util.NotEmptyValidator"
      txt=" Database Hostname is a required field" />
 </field>
 ...
 </panel>
```

该面板的 order 属性被设置为 0 ，这对应于 清单 2中的面板集合中定义的 UserInputPanel 的数量。

在用户输入面板中，可以为用户定义信息文本。而且，我加入了一个 NotEmptyValidator ，它要求用户在文本域中输入一个值。这样可以防止因用户忘记输入必需的信息而导致安装错误。图 3 中显示了基于 清单 3的 UserInputPanel ：

图 **3.** 供用户输入信息的面板



用户常常根据输入信息和显示消息的难易程度来判断安装程序的好坏。因此一定要让用户看到的东西容易使用。

pack

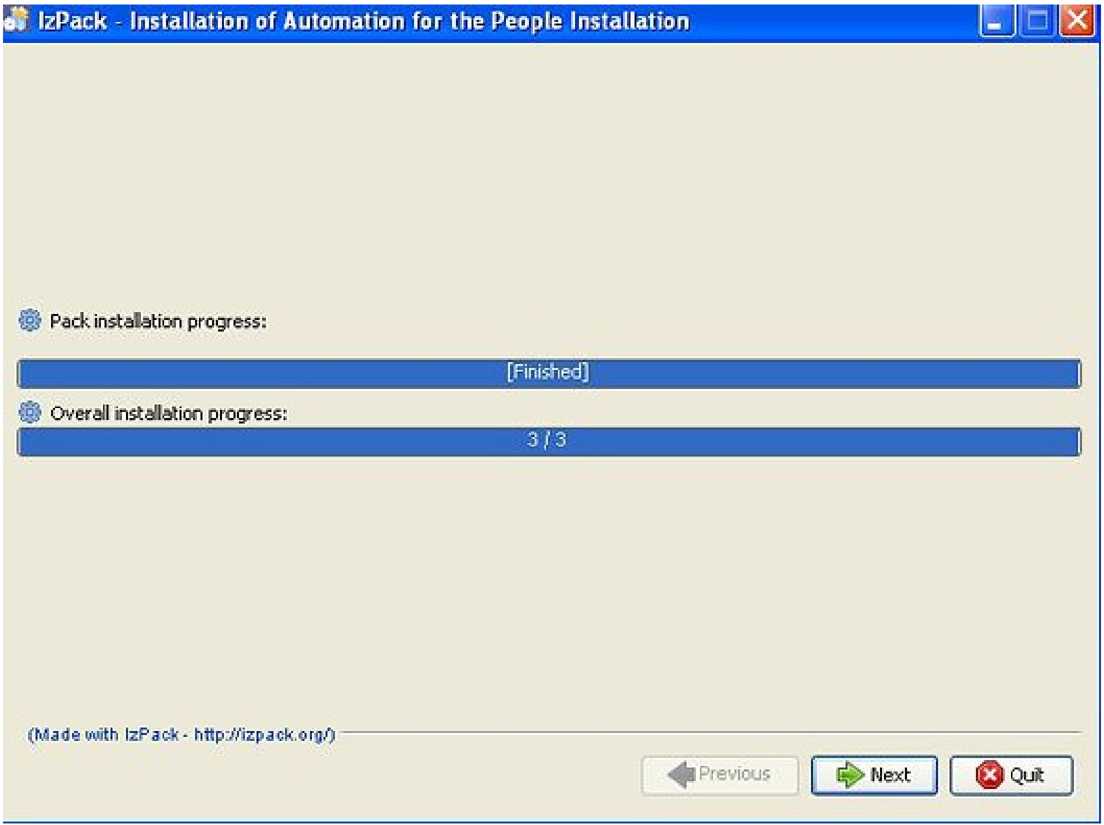
IzPack 使用术语 *pack*表示负责实际安装开发团队已实现的软件的组件。所有其他的 IzPack 组件（面板、用户输入、验证器等）都是为运行这些 pack 做准备。我的项目使用 pack 做两件事：下载我们已经用 Ant 编写好的一个 ZIP 安装发布包，然后运行这个安装。这种方法使我们可以重用之前编写的在命令行运行的命令。清单 4 定义了一个 <pack> ：

清单 **4.** 在 **install.xml** 中定义一个 **<pack>**

```
<packs>
  <pack name="download_install" id="download_install"
    installGroups="ap" required="no">
    <description>The base files</description>
    <file src="autopeople.zip.file"
      targetdir="$SYSTEM_user_home/_cnnew1_cnnew1@{installer.dir}"/>
    <file src="build.xml"
      targetdir="$SYSTEM_user_home/@{installer.dir}"/>
    <file src="property-template"
      targetdir="$SYSTEM_user_home/@{installer.dir}">
      <excludes>**/.svn/**</excludes>
    </file>
  </pack>
```

图 4 显示一个进行中的 pack 安装：

图 **4.** 执行一个 **pack**



IzPack 中的 packs 可以包含多个 *pack*！如果您已经完成了用户验证、诊断和获取特定于环境的信息等前期工作，那么用户应该很容易运行安装包。

运行 Ant 脚本

在我的团队中，我们花了很多时间用 Ant 创建基于发布包的安装程序。我们不想在 IzPack 中再次重新实现该功能。幸运的是，IzPack 支持调用已有 Ant 文件。还记得吗，我在 清单 1中定义资源时，曾列出了 antActionSpec.xml 作为一个资源。清单 5 显示了摘自 antActionSpec.xml 脚本的一个片段：

清单 5. 在 antActionSpec.xml 中执行 pack 行为

```
<antactions>
 <pack name="download_install">
   <antcall buildfile="$SYSTEM_user_home/${installer.dir}/build.xml"
     order="afterpack"
     verbose="yes"
     logfile="$SYSTEM_user_home/${installer.dir}/antlog_installer.txt"
     inheritall="false"
     messageid="AntAction.download-install">
     <target name="install"/>
     <property name="install.path" value="$SYSTEM_user_home/${installer.dir}"/>
    </antcall>
 </pack>
...
</antactions>
```

这个脚本中最重要的执行 `build.xml` 的部分。这是现有的 Ant 构建脚本，它执行下载和提取一个 ZIP 安装文件，安装并配置 Web 容器，然后完成安装中剩下的其他任务。antActionSpec.xml 使我们可以重用现有的 Ant 脚本。

编译安装程序

最后一步就是 IzPack 的 编译。编写好 install.xml 和相关脚本之后，就可以生成安装程序。清单 6 是一个可用于生成 install.jar（可以修改这个文件的文件名）的单行命令的例子：

清单 **6.** 创建一个安装程序

```
compile ../sample/install.xml -b ../sample
```

清单 6 中的命令假设您是从 IzPack 的 bin 子目录运行它。 `sample` 是对 IzPack 提供的 sample 子目录的引用。生成安装程序后，可以通过从 sample 子目录中生成 install.jar 的位置运行 `java -jar install.jar` 来测试它。

结束语

在本文中，我展示了如何使用 IzPack 的不同组件为用户创建易于使用的安装包。他们可能是安装基于客户机的软件的用户，也可能是安装和配置多个服务器的远程站点的用户，还可能是安装和配置企业工具套件的团队。如果软件容易安装，则更易被采纳，这一点在安装比较复杂的环境中尤为突出。如果安装需要很多手动步骤或者干脆无法进行，那么用户很快就会对软件失去信心。通过 IzPack 等工具使安装变得更加容易，这可以帮助您赢得并留住热情的用户。

让开发自动化: 针对广大开发人员的并行开发

Subversion 中的分支、标记和合并

虽然很多开发团队都使用版本控制系统管理代码变更，但当多个开发人员并行地使用不同的代码库进行编码时，还是会出现问题的。在本期的 让开发自动化 中，自动化专家 Paul Duvall 展示了如何运用开源的、免费的 Subversion 版本控制系统来有效地进行标记、分支和合并。

说到源代码分支，可以将大多数的软件开发团队大致划分为两大阵营：有些是根本不分支；或存在大量的分支（甚至储存库），以致开发人员不知道从哪里签入变更 — 或者觉得合并变更很痛苦，于是就冒险将这项工作推迟到软件快要发布时才做。

术语

主干（*trunk*）（有时称 *head*）用于干线开发（mainline development）。分支（*branch*）是指一个代码支线副本，用于进行与干线开发不同的变更。标记（*tag*）（有时称 标签）是一个使用时间戳的代码支线副本，用于标识代码支线，以在开发周期中返回到标记的地方。

永远只需要操作主干是最理想的情况。这使合并两个或多个代码支线间的变更没有那么复杂。然而，在现实的软件开发中，您正在开发的可能是未来版本，或者有时您可能需要为一个已经交付使用的版本准备一条后路。你需要有权限访问已发布版本的源代码副本 — 但又不能扰乱正在开发的新代码。

但当开发团队试图使用分开的代码支线时，问题就会出现了。有些时候，开发团队可能会选择不创建分支，免得会延误发布或导致开发人员瓶颈。而有些时候，开发人员合并的频率太低，结果导致了合并冲突、瓶颈以及发布延误。而增加分支则会使导航项目储存库很困难，从而导致开发人员无意中更改了不应该更改的代码。

团队进行并行开发时，一定要以最高的频率将代码合并回干线（即主干）。如果无法经常将代码合并到主干的话，可以运行测试，这样就能够确定是否会发生合并冲突，从而使实施 合并没有那么困难。要有效地进行并行开发，您可以使用 Subversion（SVN）中的标记和分支，Subversion 是一个开源的、免费的源代码管理系统。通过标记，团队可以安全地返回到源代码的前一版本中。

我将通过介绍以下内容来示范如何在 SVN 中进行并行开发：

- 如何从主干创建一个 SVN 版本标记
- 根据版本标记来创建一个 SVN 分支
- 将变更合并回干线（即主干）的技巧
- 如何在开发中的主干运行持续集成（Continuous Integration，CI），以定期测试分支与主干的合并

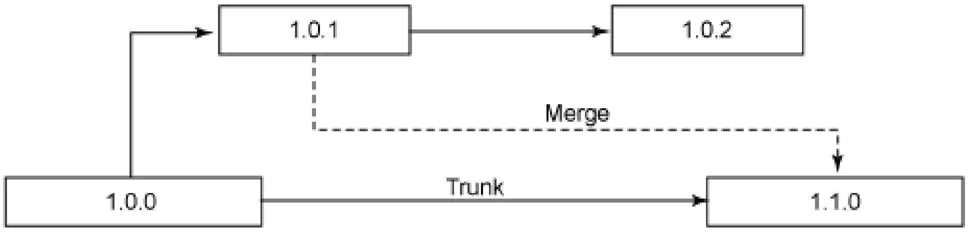
- 演示如何将源于分支的变更应用到主干
- 举例说明如何标记分支的源代码

关于本系列

作为开发人员，我们致力于自动化用户流程；但许多开发人员却疏忽了自动化自己的开发流程。为此，我们编写了 让开发自动化 系列文章，专门探讨软件开发流程自动化的实际应用，为您介绍何时 以及如何 成功应用自动化。

图 1 显示了几个并行代码支线的基本流程：

图 **1.** 并行开发



在图 1 中，有效的开发发生在 SVN 主干的版本 1.0.0 和版本 1.1.0 之间。可以是一组开发人员在版本 1.0.1 分支上进行开发，而其他人员在干线上开发。

如果想要多个开发人员负责不同的代码支线的话，可以使用很多策略和技巧。在本文中，我将展示一个很常用的方法，我曾在使用 SVN 的项目上用过它。

为并行开发配置 **Subversion**

安装和配置 SVN 服务器并不在本文的讨论范围之内。如果您有权限访问一个有效的 SVN 服务器，就可以执行以下的步骤了：

1. 将 SVN 客户机软件下载到您的工作站。
2. 在工作站中创建一个标准本地目录。
3. 将目录添加到 SVN 储存库。
4. 将目录提交到 SVN 储存库。

从 Tigris.org Web 站点（参见 参考资料）为您的操作系统下载 SVN 客户机软件，并将其安装到您的工作站。确保 SVN 可执行文件在您的工作站的系统目录中。用 `svn co _URL_` 执行储存库的 SVN 签出。

接下来，创建三个本地目录：

- 分支：用于维护干线开发之外的软件。
- 标记：在发布软件时用于标识变更集，以备使用。
- 主干：用于干线开发。

清单 1 展示了在 Windows®、Macintosh 以及基于 *nix 的系统上如何从命令行创建这些目录：

清单 **1.** 创建本地目录，将其添加到 **Subversion**

```
$ mkdir branches
$ mkdir tags
$ mkdir trunk
```

在操作系统中创建了目录之后，您可以分别使用 SVN `add` 和 `commit` 命令将它们添加并提交到 SVN。在我创建清单 1 的目录的目录中，我输入了如清单 2 所示的命令（在适当的时候替代用户凭证）：

清单 **2.** 将本地目录添加并提交到远程 **SVN** 储存库

```
$ svn add *.*
$ svn commit -m "Setting up standard SVN branches, tags and trunk directories" \
  --username tjefferson --password Mont!cello
```

执行了清单 1 和清单 2 中的操作之后，SVN 储存库应该类似于图 2：

图 **2.** 在储存库中创建的标准 **SVN** 目录



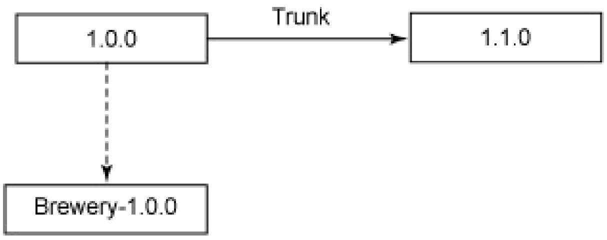
File	Extension	Revision	Author	Size	Date
https://brewery-ci.googlecode.com/svn					
branches		1			9/14/2007 8:05:20 PM
tags		1			9/14/2007 8:05:20 PM
trunk		72	pduvall		9/4/2008 9:59:23 AM

基本的 SVN 储存库就绪以后，就可以创建版本标记了。

根据主干创建一个版本标记

标记的用途是在某个特定点及时标识代码支线副本，以便以后返回到该版本。图 3 展示了一个名为 `brewery-1.0.0` 的标记，它是针对 `1.0.0` 版本创建的。（标记能够随时在任何点创建，但通常都是在发布软件时创建）。

图 **3.** 为 **SVN** 主干创建一个惟一的标记



假设主干包含已发布的软件的源代码的话，第一个任务就是要依据主干创建一个 SVN 标记。清单 3 就是一个关于如何创建这个标记的例子：

清单 **3.** 根据主干创建一个版本标记

```
<path id="svn.classpath">
  <fileset dir="${lib.dir}">
    <include name="**/*.jar" />
  </fileset>
</path>
<taskdef name="svn" classpathref="svn.classpath"
  classname="org.tigris.subversion.svnant.SvnTask"/>

<target name="create-tag-from-trunk">
  <svn username="jhancock" password="S!gnhere">
    <copy srcUrl="https://brewery-ci.googlecode.com/svn/trunk"
      destUrl="https://brewery-ci.googlecode.com/svn/tags/brewery-1.0.0"
      message="Tag created by jhancock on ${TODAY}" />
  </svn>
</target>
```

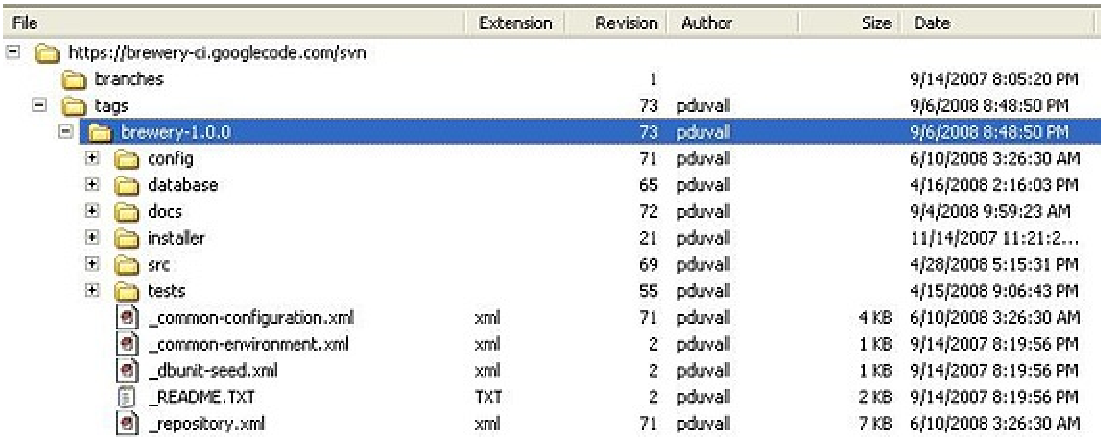
安全验证失败

第一次运行使用 Hypertext Transfer Protocol 而不是 Secure Socket Layer 的 SVN 服务器时，一定要接受安全认证。如果您是第一次这样从 Ant 脚本连接到安全 SVN 服务器，连接会失败，并且不提供诊断信息。因此，第一次时，必须从命令行运行 SVN 命令连接到服务器。以后，您就可以从您的工作站运行任何 SVN Ant 脚本，以连接到该服务器。

清单 3 使用了由 Subclipse 开源项目提供的 SVN Ant 任务（下载地址请参见 参考资料）。运行该 Ant 脚本时，一定要将随 SVN Ant 任务一起提供的 JARs —svnant.jar、svnClientAdapter.jar 和 svnjavahl.jar— 包含在您的类路径中。清单 3 的前半部分定义了这个类路径。中间部分使用 `taskdef` 定义了 SVN Ant 任务。最后，我向主干和标记目录执行了 SVN `copy` 命令，从而为这个版本提供一个惟一的名称：`brewery-1.0.0`.

运行清单 3 中的脚本并创建了一个新标记之后，您的 SVN 储存库应该如图 4 所示。储存库的根级下面是标记目录（在 清单 2 中创建）。而标记目录的下面是在清单 3 中创建的新标记（目录）：brewery-1.0.0。它含有主干的副本。

图 **4.** 根据主干创建标记



File	Extension	Revision	Author	Size	Date
https://brewery-ci.googlecode.com/svn					
branches		1			9/14/2007 8:05:20 PM
tags		73	pduvall		9/6/2008 8:48:50 PM
brewery-1.0.0		73	pduvall		9/6/2008 8:48:50 PM
config		71	pduvall		6/10/2008 3:26:30 AM
database		65	pduvall		4/16/2008 2:16:03 PM
docs		72	pduvall		9/4/2008 9:59:23 AM
installer		21	pduvall		11/14/2007 11:21:2...
src		69	pduvall		4/28/2008 5:15:31 PM
tests		55	pduvall		4/15/2008 9:06:43 PM
_common-configuration.xml	xml	71	pduvall	4 KB	6/10/2008 3:26:30 AM
_common-environment.xml	xml	2	pduvall	1 KB	9/14/2007 8:19:56 PM
_dbunit-seed.xml	xml	2	pduvall	1 KB	9/14/2007 8:19:56 PM
_README.TXT	TXT	2	pduvall	2 KB	9/14/2007 8:19:56 PM
_repository.xml	xml	71	pduvall	7 KB	6/10/2008 3:26:30 AM

虽然标记的内容在 Subversion 中是可以更改的，但千万 不要这样做。

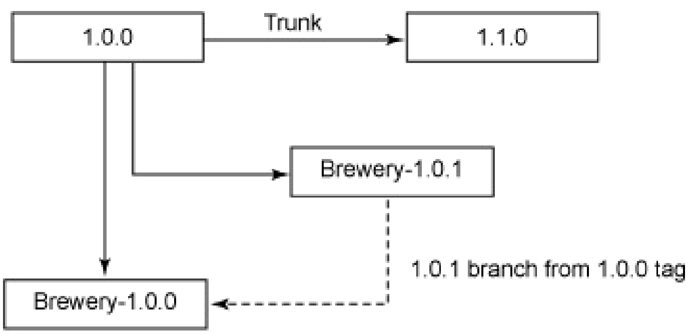
根据版本标记创建一个分支

在技术上，根据版本标记创建分支与根据主干创建标记是相似的。两者都要使用到 SVN 的 `copy` 命令。通常都会依照标记而创建分支，因为标记的是已发布 的代码的代码副本 — 而不是正在开发的代码（可能已经更改）。图 5 展示了如何根据 `1.0.0` 版本标记创建 `1.0.1` 分支：

版本命名

人们一直都想找到一个简单的版本命名的模式。但是，只要一个版本和下一个版本或不同项目之间所使用的版本命名模式稍有不同的话，麻烦就大了。版本模式可能有很多种。一定要选择一个既简单又能够灵活处理将来的版本的模式。我所使用的模式是 major-version.minor-version.patch，它很简单。Version 1.1.2 就是以这个命名模式命名的。也有些团队选择向版本追加一个构建号。

图 **5.** 根据 **1.0.0** 版本标记创建分支 **1.0.1**



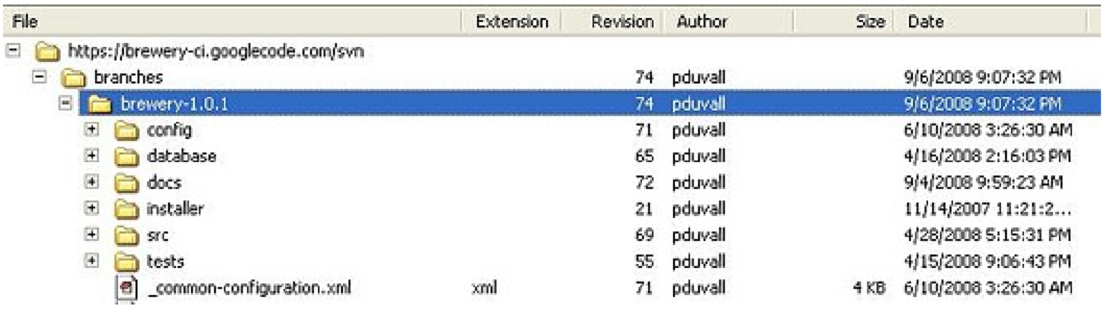
清单 4 通过 SVN Ant 任务调用了 SVN `copy` 命令，以将 `brewery-1.0.0` 标记中的所有文件拷贝到分支位置：

清单 **4.** 从版本标记创建分支的 **Ant** 脚本

```
<target name="create-branch-from-tag">
  <svn username="sadams" password="b0stonM@ss">
    <copy srcUrl="https://brewery-ci.googlecode.com/svn/tags/brewery-1.0.0"
      destUrl="https://brewery-ci.googlecode.com/svn/branches/brewery-1.0.1"
      message="Branch created by sadams on ${TODAY}" />
  </svn>
</target>
```

运行清单 4 中的脚本之后，SVN 储存库应该如图 6 所示：

图 **6.** 从版本标记创建分支



记住主干

有些团队处理分支开发时很极端，他们所有的开发工作都从分支开始。要记住，根据主干进行开发会更轻松、更易于管理。分支就是为了并行地进行分开的开发。尽量不要滥用分支，动不动就创建分支。

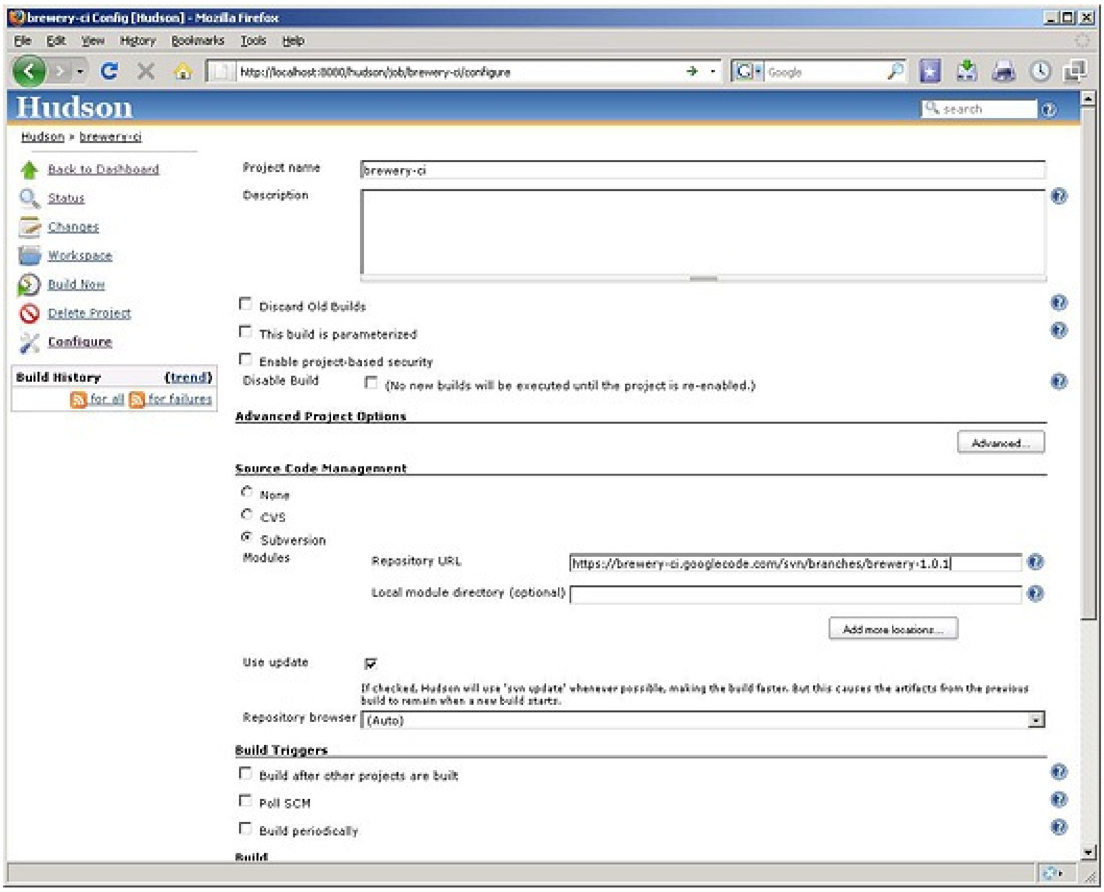
在创建分支和使用 SVN Ant 任务时，一定要记得使用标记，这样您才能够提供一个可重复的过程，该过程方便源代码的维护，使您可以轻松返回到上一版本的源代码。

根据分支运行 CI

通常 CI 过程都是根据储存库的干线（即主干）而运行的。但如果想集成开发人员在分支上的变更并检查支线与干线的合并，这个原理也适用于分支。

图 7 展示了 SVN 的位置。在这个 Hudson 配置页面上，您还能够定义要调用的 Ant 目标。

图 **7. Hudson CI** 服务器根据主干构建分支并测试合并

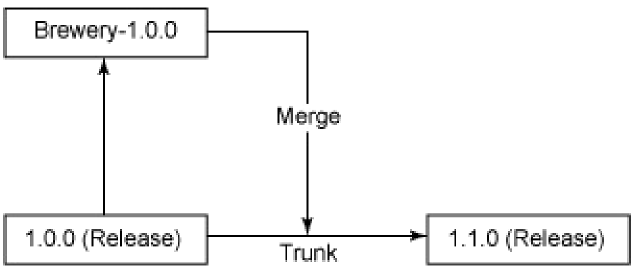


运行 Hudson 等 CI 服务器来测试合并可以提供一个预警系统，警告可能会在开发周期中发生潜在的合并冲突。

将分支的变更合并到主干

创建分支的主要原因之一就是防止中断干线开发。但是，一定要将分支上的更改合并到主干。图 8 展示了将版本 1.0.1 合并到干线，这个干线是软件正在开发的版本 1.1.0：

图 **8. SVN** 时间线



在清单 5 中，我使用了 Subversion 的 `merge` 命令。我先输入 `svn merge`，接着是合并到的目标 URL，然后是合并源的 URL，最后是本地目录位置：

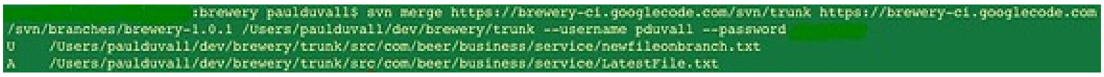
清单 **5.** 使用 **SVN** 的 **`merge`** 命令将分支开发合并到主干

```
$ svn merge https://brewery-ci.googlecode.com/svn/trunk \
 https://brewery-ci.googlecode.com/svn/branches/brewery-1.0.1 \
 /dev/brewery --username pduvall --password password!
```

SVN Ant 任务没有提供合并命令，因此需要从命令行运行 `merge` 命令。或者使用 Ant 的 `exec` 任务来运行它。

运行清单 5 中的命令会得到类似于图 9 的结果：

图 **9.** 将分支合并到主干的结果



如果合并成功，则需要提交 Subversion 中的变更，如清单 6 所示。在命令行上输入 `svn commit` ，然后输入消息描述和主干的 SVN URL ：

清单 **6.** 将合并后的变更提交到主干

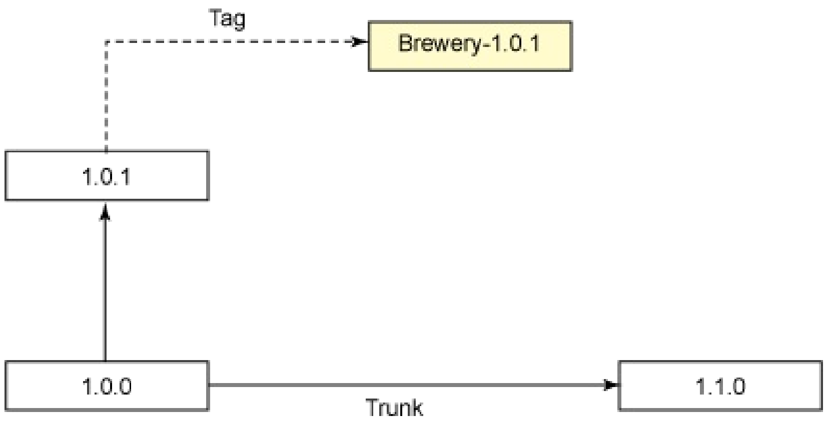
```
<target name="commit-branch-to-trunk">
  <svn username="gwbush" password="IL0veTHEG00g!e">
    <commit dir="${basedir}"
      message="Committing changes from brewery-1.0.1">
    </commit>
  </svn>
</target>
```

要经常将变更从分支合并到主干以避免合并冲突，从而使代码支线始终保持一致。

根据分支创建标记

为了根据特定的分支准备一个版本，我创建了一个 SVN 标记。这里使用方法与前面一些清单提供的方法类似。图 10 展示了根据 `brewery-1.0.1` 分支创建名为 `brewery-1.0.1` 的标记的方法：

图 **10.** 根据分支创建标记



在特定的分支上完成开发后，就需要在 Subversion 中标记它。清单 7 展示了一个根据分支创建标记的例子：

清单 **7.** 根据分支创建 **SVN** 标记

```
<svn username="jbartlett" password="newHampsh!re">
  <copy srcUrl="https://brewery-ci.googlecode.com/svn/branches/brewery-1.0.1"
    destUrl="https://brewery-ci.googlecode.com/svn/tags/brewery-1.0.1"
    message="Branch created by jbartlett on ${TODAY}" />
</svn>
```

根据具体分支创建标记之后，您就可以在以后的开发周期中返回到该版本了。

结束语

并行开发并不是什么难事，但如果不按照项目需求进行规划并不断改进，它的管理将非常困难。如果必须记住一点的话，那就是一切最终都要回到主干。您可以将分支看作是一个容纳可能中断干线开发的源代码的临时居所。最后要考虑的是要尽早地、经常地测试合并。当然可能有比 Subversion 更好的支持并行开发的版本控制系统，但根据我的经验，开发团队在开发时坚持原则比使用工具解决问题重要得多。

让开发自动化: 实现自动化数据库迁移

使用 *LiquiBase* 管理数据库变更

数据库通常不能够与它们支持的应用程序保持同步，从管理方面来讲，将数据库和数据置于一个已知状态是个很大的挑战。在本期的 让开发自动化中，自动化专家 Paul Duvall 演示了如何使用开源的 LiquiBase 数据库迁移工具轻松地处理数据库和应用程序的频繁变更。

在过去几年中，我使用过的大多数应用程序都是需要管理大量数据的企业应用程序。从事这类项目的开发团队常常将数据库视为与应用程序完全脱离的单独实体。造成这种现象的原因是组织结构经常将数据库团队从应用程序开发团队分离出来。有时候，这是团队的习惯引起的。不管怎样，我发现这种分离会导致（或忽略）一些实践：

- 手工变更数据库
- 不能与团队的其他成员分享数据库变更
- 使用不一致的方法变更数据库或数据
- 使用低效的手工方法管理数据库版本之间的变更

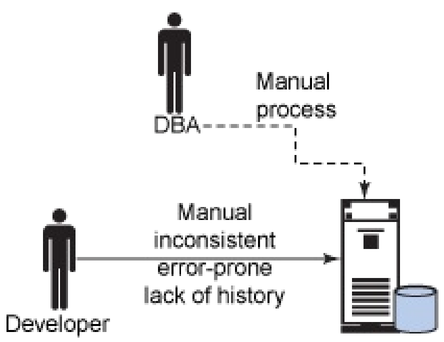
这些实践效率低下，使开发人员无法与数据变更保持同步。而且，还使应用程序的 用户遇到与数据不一致和数据损坏等问题。

图 1 演示了软件开发项目中经常用到的手工方法。手工方法在使用时通常不能保证一致性并且容易产生错误，撤销已完成的工作很困难，而且难以分析数据库变更的历史。例如，某个 DBA 可能想变更查找数据，但是开发人员却忘记将这个数据插入到同一个表中。

关于本系列

作为开发人员，我们致力于用户自动化流程；但许多开发人员却疏忽了自动化自己的开发流程。为此，我们编写了 让开发自动化系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍 何时以及 如何成功应用自动化。

图 **1.** 手工变更数据库

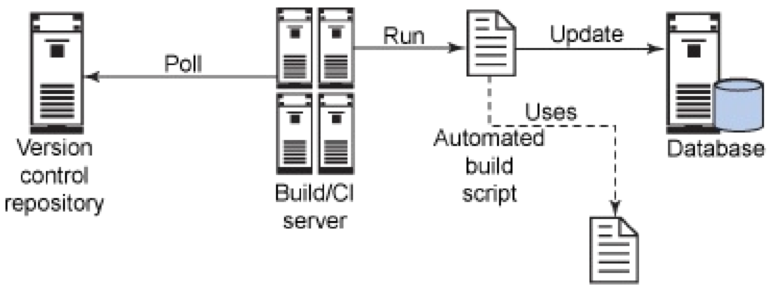


通过实现最小化人为干预的数据库变更策略，可以避免手工方法带来的缺陷。通过结合各种实践和工具，可以使用一致且可重复的过程变更数据库和数据。在本文中，我将介绍以下内容：

- 使用一种称为 LiquiBase 的工具在各个数据库版本之间进行迁移
- 如何自动运行数据库迁移
- 一致地变更数据库的实践
- 使用 LiquiBase 进行数据库重构

在图 2 中，一个 Build/Continuous Integration 服务器轮询版本控制库（例如子版本）中的变更。当它发现一个变更后，将运行一个自动化构建脚本，该脚本使用 LiquiBase 更新数据库。

图 **2.** 自动化数据库迁移



通过使用类似图 2 所示的过程，团队中的任何人都可以将相同的变更应用到数据库中 —可以是本地或共享数据库服务器。此外，由于这个过程使用了自动化脚本，因此这些变更不需要任何人为干预就可以应用到不同环境中。

DDL、DML 模式、数据库或数据

在本文中，我使用术语 数据库变更表示通过应用数据定义语言（Data Definition Language，DDL）脚本变更数据库结构。（一些数据库供应商将之称为 模式）。同时，我将通过数据定义语言（DML）脚本变更数据库称为 数据变更。

使用 LiquiBase 管理数据库变更

LiquiBase（从 2006 年开始投入使用）是一种免费开源的工具，可以实现不同数据库版本之间的迁移（参见 参考资料）。目前也存在少量其他开源数据库迁移工具，包括 openDBcopy 和 dbdeploy。LiquiBase 支持 10 种数据库类型，包括 DB2、Apache Derby、MySQL、PostgreSQL、Oracle、Microsoft®SQL Server、Sybase 和 HSQL。

要安装 LiquiBase，下载经过压缩的 LiquiBase Core 文件，解压缩，然后将包含的 liquibase-*version*.jar 文件放到系统路径中。

要开始使用 LiquiBase，需要以下四个步骤：

1. 创建一个数据库 变更日志（*change log*）文件。
2. 在变更日志文件内部创建一个 变更集（*change set*）。
3. 通过命令行或构建脚本对数据库运行变更集。
4. 检验数据库中的变更。

创建一个变更日志和变更集

要运行 LiquiBase，如清单 1 所示，首先要创建一个 XML 文件，也称为数据库变更日志：

清单 **1.** 在 **LiquiBase XML** 文件中定义一个变更集

```
<?xml version="1.0" encoding="UTF-8"?>

<databaseChangeLogxmlns="http://www.liquibase.org/xml/ns/dbchangelog/1.7"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog/1.7
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-1.7.xsd">
 <changeSetid="2" author="paul">
   <createTable tableName="brewer">
     <column name="id" type="int">
       <constraints primaryKey="true" nullable="false"/>
     </column>
     <column name="name" type="varchar(255)">
       <constraints nullable="false"/>
     </column>
     <column name="active" type="boolean" defaultValue="1"/>
   </createTable>
 </changeSet>
</databaseChangeLog>
```

再次使用 **XML**

有些开发人员会使用 XML，而其他人则没有涉足这个领域。很多开发人员甚至已经习惯使用 XML 脚本进行编程（例如，使用 Apache Ant），但是 DBA 不一定用到。最近，我非常兴奋地向一个 DBA 同事展示 LiquiBase 的一些特性。他非常喜欢其中一些强大的数据库变更管理工具，但是很怀疑 DBA 是否会使用基于 XML 的语法。我向他保证，LiquiBase 还通过它的 `sqlFile` 和 `sql` 定制重构支持调用定制 SQL 脚本。

可以看到，数据库变更日志文件包括一个 XML 模式引用（LiquiBase 安装中包含的 dbchangelog-1.7.xsd 文件）。我在变更日志文件中创建了一个 `<changeSet>` 。在 `<changeSet>` 中，我使用结构化的方式将变更应用到数据库，如 LiquiBase 模式所定义。

从命令行运行 **LiquiBase**

定义完变更集后，可以从命令行运行 LiquiBase，如清单 2 所示：

清单 **2.** 从命令行运行 **LiquiBase**

```
liquibase --driver=org.apache.derby.jdbc.EmbeddedDriver \
--classpath=derby.jar \
--changeLogFile=database.changelog.xml\
--url=jdbc:derby:brewery;create=true \
--username= --password= \
update
```

在本例中，运行 LiquiBase 传入的内容：

- 数据库驱动器
- 数据库驱动器 JAR 文件的位置所在的类路径
- 所创建的变更日志文件（如 清单 1所示）名称为 database.changelog.xml
- 数据库的 URL
- 用户名和密码

最后，清单 2 调用 `update` 命令告诉 LiquiBase 将变更应用到数据库中。

在自动构建中运行 LiquiBase

这里并不使用命令行选项，通过调用 LiquiBase 提供的 Ant 任务，可以将数据库变更作为自动化构建的一部分。清单 3 展示了 Ant 任务的示例：

清单 **3.** 执行 **`updateDatabase`** **Ant** 任务的 **Ant** 脚本

```
<target name="update-database">
 <taskdef name="updateDatabase" classname="liquibase.ant.DatabaseUpdateTask"
   classpathref="project.class.path" />
 <updateDatabase changeLogFile="database.changelog.xml"
   driver="org.apache.derby.jdbc.EmbeddedDriver"
   url="jdbc:derby:brewery"
   username=""
   password=""
   dropFirst="true"
   classpathref="project.class.path"/>
</target>
```

在清单 3 中，创建了一个名为 `update-database` 的目标。在其中定义了一个将要用到的特殊 LiquiBase Ant 任务，称为 `updateDatabase`。我传入需要的值，包括 `changeLogFile`（指定清单 1中定义的变更日志文件）和数据库的连接信息。`classpathref` 中定义的类路径必须包含 liquibase-*version*.jar。

运行前后

图 3 展示了在 清单 1中运行变更集之前的数据库状态：

图 **3.** 运行 **LiquiBase** 变更集之前的数据库状态

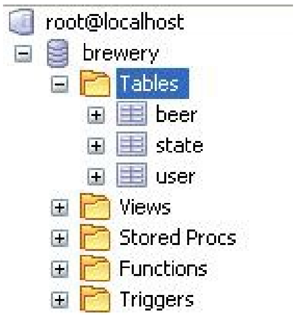
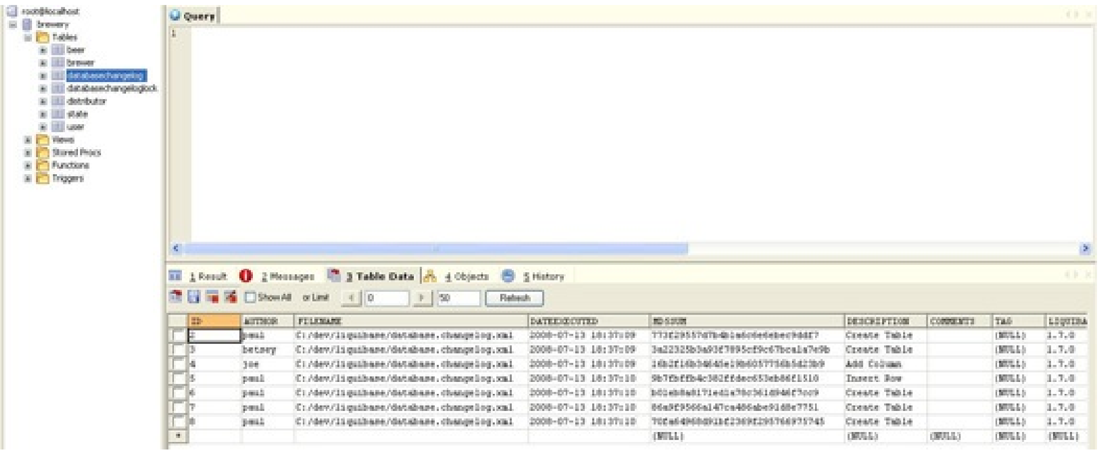


图 4 展示了运行数据库变更集的结果，可以通过命令行（如 清单 2所示）或从 Ant（如 清单 3所示）运行：

图 **4.** 运行 **LiquiBase** 变更集后将变更应用到数据库



查看 完整的图。

需要注意图 4 中的几个方面。创建了两个特定于 LiquiBase 的表，以及一个根据 清单 1中的变更集定义创建的新表。第一个特定于 LiquiBase 的表称为 `databasechangelog`，它跟踪应用到数据库的所有变更 —有助于跟踪谁执行了数据库变更以及原因。第二个特定于 LiquiBase-的表是 `databasechangelock`，标识出具有数据库变更锁的用户。

还可以使用多种其他方式运行 LiquiBase，但我已经介绍了应用数据库变更所需的大部分信息。在使用 LiquiBase 时，将花很多时间研究应用数据库重构的各种方法，以及变更特定数据库的复杂性。例如，LiquiBase 提供了数据回滚支持，这可能是个很大的挑战。在展示数据库重构示例之前，我将快速浏览一些数据库集成的基本原则和实践，它们能帮助您充分利用数据库迁移。

频繁集成数据库变更

最近几年，开发团队将类似于处理源代码的原则和实践应用到数据库资产管理中。因此，可以将数据库变更编写为脚本、在一个源代码库中共享这些资产，以及将变更集成到构建和持续集成过程，这只是自然的演进。表 1 概括了开发团队将数据库变更变成一个自动化过程的

一部分时，需要遵循的关键实践：

自动化 DBA

在我曾经参与的一些项目中，DBA 在控制开发数据库的变更时造成了一些不必要的瓶颈。DBA 应该把时间花在一些创新的、非重复性行为，例如监视和改善数据库性能，而不是假借控制性和一致性的名义做一些无用的重复性工作。

表 1. 数据库集成实践

实践	说明
脚本化所有 DDL 和 DML	数据库变更应该能够从命令行运行。
数据资产的源代码控制	使用一个版本控制库管理所有与数据库相关的变更。
本地数据库沙盒	每个开发人员使用一个本地数据库沙盒执行变更。
自动化数据库集成	将数据库相关的变更作为构建过程的一部分。

这些实践确保了更好的一致性并防止变更在软件版本转换之间丢失。

对现有数据库应用重构

随着新特性添加到了应用程序中，经常需要变更数据库的结构或修改表约束。LiquiBase 提了超过 30 种数据库重构支持（参见 参考资料）。本节将介绍 4 种重构：添加列（Add Column）、删除列（Drop Column）、创建表（Create Table）和操作数据。

添加列

在项目的开始，几乎不可能考虑到数据库中的所有列。而有时候，用户要求新的特性 —例如为存储在系统中的信息收集更多的数据 —这就要求添加新的列。清单 4 使用 LiquiBase `addColumn` 重构，向数据库中的 `distributor` 表添加了一个列：

清单 4. 使用 **LiquiBase** 变更集中的 **Add Column** 数据库重构

```
<changeSet id="4" author="joe">
 <addColumn tableName="distributor">
   <column name="phonenumber" type="varchar(255)"/>
 </addColumn>
</changeSet>
```

新的 `phonenumber` 列被定义为 `varchar` 数据类型。

删除列

假如在以后几个版本中，您想要删除在清单 4 添加的 `phonenumber` 列。只需要调用 `dropColumn` 重构，如清单 5 所示：

清单 **5.** 删除一个数据库列

```
<dropColumn tableName="distributor" columnName="phonenumber"/>
```

创建表

向数据库添加一个新表也是常见的数据库重构。清单 6 创建了一个新表 `distributor`，定义了列、约束和默认值：

清单 **6.** 在 **LiquiBase** 中创建一个新数据库表

```
<changeSet id="3" author="betsey">
 <createTable tableName="distributor">
   <column name="id" type="int">
     <constraints primaryKey="true" nullable="false"/>
   </column>
   <column name="name" type="varchar(255)">
     <constraints nullable="false"/>
   </column>
   <column name="address" type="varchar(255)">
     <constraints nullable="true"/>
   </column>
   <column name="active" type="boolean" defaultValue="1"/>
 </createTable>
</changeSet>
```

这个示例使用了 `createTable` 数据库重构作为变更集的一部分（清单 1中也使用了 `createTable`）。

操作数据

在应用了结构性数据重构后（例如添加列和创建表），通常需要向受重构影响的表中插入数据。此外，可能需要修改查找表（或其他类型的表）中的现有数据。清单 7 展示了如何使用一个 LiquiBase 变更集插入数据：

清单 **7.** 使用一个 **LiquiBase** 变更集插入数据

```
<changeSet id="3" author="betsey">
 <code type="section" width="100%">
 <insert tableName="distributor">
   <column name="id" valueNumeric="3"/>
   <column name="name" value="Manassas Beer Company"/>
 </insert>
 <insert tableName="distributor">
   <column name="id" valueNumeric="4"/>
   <column name="name" value="Harrisonburg Beer Distributors"/>
 </insert>
</changeSet>
```

您应该编写用于操作数据的 SQL 脚本，因为使用 LiquiBase XML 变更集限制很多。有时候使用 SQL 脚本向数据库应用大量的变更会简单一些。LiquiBase 也可以支持这些情景。清单 8 调用变更集中的 `insert-distributor-data.sql` 来插入 `distributor` 表数据：

清单 **8.** 从 **LiquiBase** 变更集运行一个定制 **SQL** 文件

```
<changeSet id="6" author="joe">
 <sqlFile path="insert-distributor-data.sql"/>
</changeSet>
```

LiquiBase 支持很多其他数据库重构，包括 Add Lookup Table 和 Merge Columns。可以使用如清单 4 到清单 8 所示的方式定义所有这些支持。

持续保持数据同步

在软件开发中，如果遇到一些难题，您应该更多地关注它，而不是等到以后才手动执行这些操作，从而使问题变得更严重，花费也更大。数据库迁移非常重要，自动化迁移过程能够获得很多好处。在本文中，我已经介绍了以下内容：

- 演示如何使用 LiquiBase 脚本化数据库迁移并将这些变更变成自动化构建过程的一部分
- 描述了实现一致性的数据库集成的原则和实践
- 展示如何通过使用 LiquiBase 应用数据库重构，比如 Add Column、Create Table 和更新数据

表 2 总结了 LiquiBase 提供的一些特性的列表：

表 **2. LiquiBase** 部分特性总结

特性	说明
支持多个数据库	支持 DB2、Apache Derby、MySQL、PostgreSQL、Oracle、Microsoft SQL Server、Sybase 和 HSQL 等。
查看应用到数据库的变更的历史。	使用 `databasechangelog` 表，可以查看应用到数据库的每一个变更。
生成数据库差异日志	了解 LiquiBase 变更集以外的应用到数据库的变更。
能够运行定制 SQL 脚本	使用 LiquiBase 调用已经编写好的 SQL 脚本。
回滚数据库变更的工具	可以对应用到数据库的任何变更执行回滚。

可以看到，通过自动化脚本恰当地应用变更时，数据库迁移变得更加轻松，并且成为团队中的多数成员都可以运行的重复过程。

让开发自动化: 持续重构

使用静态分析工具识别代码味道

重构是公认的改进现有代码的好方法。然而，如何通过一种一致且可重复的方式 找到需要重构的代码呢？本期的 让开发自动化阐述了如何使用静态分析工具来识别需要重构的代码味道，并举例说明了如何改进坏味道代码。

在过去的几年里，我曾看过很多项目的大量源代码，从精美的设计到像是用胶带绑定到一起的代码。我写过新的代码也维护过其他开发人员的源代码。我喜欢编写新的代码，但也喜欢采用一些现有的代码，以某种方法将其简化或将重复的代码提取到一个公共类中。在我早期的工作生涯中，许多人都认为如果不编写新的代码就不会有好的效率。幸好，在 20 世纪 90 年代末，Martin Fowler 编写了 *Refactoring*一书（参见 参考资料），它使得在不改变外部行为的前提下改进现有代码成为可能。

我在 本系列中所一直推崇的就是 效率：如何减少耗时过程的冗余度，更快速地执行它们。在本文的任务中，我一样推崇这个目标，并且将论述怎样更 有效地执行它们。

关于本系列

作为开发人员，我们致力于为用户自动化流程；但许多开发人员疏忽了自动化我们自己的开发流程的机会。为此，我们编写了 让开发自动化系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍 何时以及 如何成功应用自动化。

重构的一个典型方法是在引入新代码或更改方法时对现有代码做出小小的变动。该技巧面临的挑战在于一个开发团队的开发人员的应用方法不一致，并且很容易错失重构的机会。这也正是我提倡使用静态分析工具识别编码违规的原因所在。有了这些工具，您就能够从总体上了解代码库，并且处于类或方法的级别。幸运的是，在 Java™程序设计中，您可以选择的可免费下载的开源静态分析工具很多：CheckStyle、PMD、FindBugs、JavaNCSS、JDepend 等等。

在本文中，您将学习如何：

- 使用 CheckStyle 度量 圈复杂度（**cyclomatic complexity**），并提供诸如 **Replace Conditional with Polymorphism**之类的重构，以此来减少 条件复杂度代码味道
- 使用 CheckStyle 评估 代码重复率，并提供诸如 **Pull Up Method**之类的重构，以此来移除 重复代码
- 使用 PMD（或 JavaNCSS）计算 源代码行，并提供诸如 **Extract Method**之类的重构，以此来淡化 大类代码味道
- 使用 CheckStyle（或 JDepend）确定一个类的 传出耦合度（**efferent coupling**），并提供诸如 **Move Method**之类的重构，以此来除掉 过多的导入代码味道

我将使用如下的通用格式来检查每一种代码味道：

1. 描述可以指示出代码里面的问题的味道
2. 定义可以找到该味道的度量方法
3. 展示可以度量代码味道的工具
4. 提供用于修复代码味道的重构和模式（在某些情况下）

实质上，这个方法提供了一个找到和修复整个代码库中的代码味道的一个框架。这样您就可以更好地了解到代码库中较危险的部分，然后再做出更改。更好的是，我还会向您展示如何将这个方法集成到自动构建中。

您的代码有 味道么？

所谓代码味道其实只是一种 提示，提示一些内容可能存在错误。和模式类似，代码味道提供了一个通用词汇表，您可以用它来快速识别这些类型的潜在问题。在文章中 真实地示范代码味道是很有难度的，因为它可能包括很多行代码，这样就过分地加大了文章的篇幅。因此，我会只针对其中的一些味道进行示范，然后您就可以根据查看特定代码味道的经验进行推断，识别出剩余的代码味道。

条件复杂度

味道：条件复杂度

度量：圈复杂度

工具：CheckStyle、JavaNCSS 以及 PMD

重构：Replace Conditional with Polymorphism、Extract Method

味道

条件复杂度可以以几种不同的方式出现在源代码中。这种代码味道的一个例子就是含有多个条件语句，如 `if` 、`while` 或者 `for` 语句。另一种条件复杂度是以 `switch` 语句的形式呈现出来的，如清单 1 所示：

清单 **1.** 使用 **`switch`** 语句来执行条件行为

```
...
switch (beerType) {
 case LAGER:
   System.out.println("Ingredients are...");
    ...
    break;
 case BROWNALE:
   System.out.println("Ingredients are...");
    ...
    break;
 case PORTER
   System.out.println("Ingredients are...");
    ...
    break;
 case STOUT:
   System.out.println("Ingredients are...");
    ...
    break;
 case PALELAGER:
   System.out.println("Ingredients are...");
    ...
    break;
 ...
 default:
   System.out.println("INVALID.");
    ...
    break;
}
...
```

`switch` 语句本身并没有不妥。但当一个语句包含太多的选择和代码时，它就可能暗示有需要重构的代码。

度量

要确定条件复杂度代码味道，需要确定方法的 圈复杂度。圈复杂度是一种度量方法，由 Thomas McCabe 于 1975 年定义。圈复杂度数（Cyclomatic Complexity Number，CCN）度量一个方法中某一路径的数量。无论一个方法中有多少条路径，它的起始 CNN 都从 1 开始。每一个条件构造，如 `if` 、 `switch` 、 `while` 和 `for` 语句，都被分配一个 1 値和异常路径。一个方法的总的 CCN 表明了它的复杂度。很多人认为当 CCN 为 10 或超过 10 时，就表明该方法过于复杂。

工具

CheckStyle、JavaNCSS、以及 PMD 都是度量圈复杂度的开源工具。清单 2 展示了用 XML 定义的 CheckStyle 规则文件的一个代码片断。 `CyclomaticComplexity` 模块定义了一个方法的 CCN 的最大限度。

清单 **2.** 配置 **CheckStyle**，查找圈复杂度为 **10** 或大于 **10** 的方法

```
<module name="CyclomaticComplexity">
 <property name="max" value="10"/>
</module>
```

用清单 2 的 CheckStyle 规则文件、清单 3 的 Gant 例子来示范如何将 CheckStyle 作为一个自动构建的一部分来运行。（参见 什么是 Gant ？侧边栏）：

清单 **3.** 使用 **Gant** 脚本来执行 **CheckStyle** 检查

```
target(findHighCcn:"Finds method with a high cyclomatic complexity number"){
 Ant.mkdir(dir:"target/reports")
 Ant.taskdef(name:"checkstyle",
   classname:"com.puppycrawl.tools.checkstyle.CheckStyleTask",
   classpathref:"build.classpath")
 Ant.checkstyle(shortFilenames:"true", config:"config/checkstyle/cs_checks.xml",
   failOnViolation:"false", failureProperty:"checks.failed", classpathref:"libdir") {
   formatter(type:"xml", tofile:"target/reports/checkstyle_report.xml")
   formatter(type:"html", tofile:"target/reports/checkstyle_report.html")
   fileset(dir:"src"){
     include(name:"**/*.java")
   }
 }
}
```

什么是 Gant ？

Gant 是一个自动构建工具，它提供了一个支持构建依赖关系的表达能力强的编程语言。开发人员利用 Groovy 编程语言的强大功能编写 Gant 脚本。由于 Gant 提供对 Ant 的 API 的完全访问，所以任何可以运行于 Ant 的东西都可以从 Gant 脚本运行。（参见 “用 Gant 构建软件” 教程，了解 Gant。）

清单 3 中的 Gant 脚本创建了图 1 中展示的 CheckStyle 报告。该图下面的部分指示出了一个方法的 CheckStyle 圈复杂度违规。

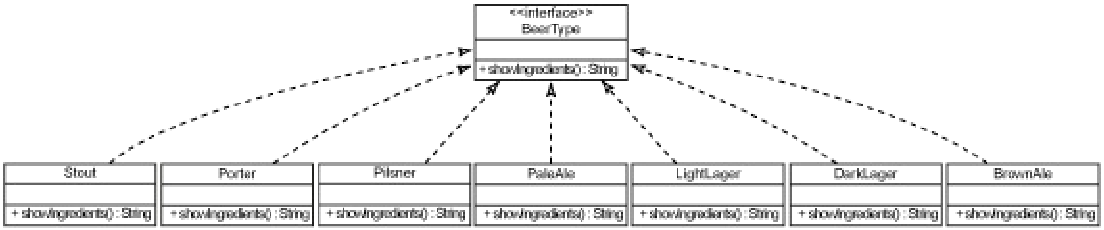
图 **1. CheckStyle** 报告根据过高的 **CCN** 来指示一种方法失败

File C:\dev\brewery-temp\src\com\beer\business\service\BeerServiceImpl.java
Error Description
Cyclomatic Complexity is 14 (max allowed is 10).
Line contains a tab character.

重构

图 2 为用 UML 表示的 *Replace Conditional with Polymorphism* 重构：

图 **2.** 用多态替代条件语句



单击 此处查看完整图。

在图 2 中，我：

1. 创建了一个叫做 `BeerType` 的 Java 界面
2. 定义了一个通用的 `showIngredients()` 方法
3. 为每一个 `BeerType` 创建了一个实现类

为了使文章保持简洁，我仅为每一个类提供一个方法的实现。显然，创建一个界面的方法可能不只一个。重构能够使代码更易于维护，如 Replace Conditional with Polymorphism 和 Extract Method（本文稍后将会讨论）。

重复代码

味道：重复代码

度量：代码重复率

工具：CheckStyle、PMD

重构：Extract Method、Pull Up Method、Form Template Method、Substitute Algorithm

味道

重复代码可能在代码库中悄然发生。有时，复制粘贴某些代码要比将该行为泛化到另一个类更简单。但复制粘贴的方法存在一个问题，即它强制将代码复制多份，并且需要维护。而且当复制出的代码发生轻微的变化而引发行为不一致时，就会发生更不易察觉的问题，具体取决于哪个方法在执行该行为。清单 4 是一个关闭代码库连接的代码示例，相同的代码出现在两种方法中：

清单 **4.** 重复代码

```
public Collection findAllStates(String sql) {
...
 try {
   if (resultSet != null) {
     resultSet.close();
   }
   if (stmt != null) {
     stmt.close();
   }
   if (conn != null) {
    conn.close();
  }
  catch (SQLException se) {
    throw new RuntimeException(se);
  }
}
...
}
...
public int create(String sql, Beer beer) {
...
 try {
   if (resultSet != null) {
     resultSet.close();
   }
   if (stmt != null) {
     stmt.close();
   }
   if (conn != null) {
    conn.close();
  }
  catch (SQLException se) {
    throw new RuntimeException(se);
  }
}
...
}
```

我有性能更好的 IDE

虽然在本文的例子中我使用 Gant 来运行查找特定味道的工具，但是使用 IDE 也同样可以解决这些问题。Eclipse IDE 就有很多静态分析工具的插件。但我仍然推荐使用自动构建工具，因为这样可以在其他环境中运行集成构建，无需使用 IDE。

度量

查找重复代码的度量方法是在代码库中的类的内部和其他类之间搜索代码重复。没有工具的话，类间的重复就更难评估。由于复制的代码通常都会发生一些轻微的变化，因此不仅要度量完全相同的代码，而且要度量 相似的代码，两者都很重要。

工具

PMD 的 Copy/Paste Detector（CPD）与 CheckStyle 这两种开源工具可以用于在整个 Java 代码库中查找相似的代码。清单 5 中的 CheckStyle 配置文件例子示范了如何使用 `StrictDuplicateCode` 模块：

清单 **5.** 使用 **CheckStyle** 找到至少 **10** 行重复代码

```
<module name="StrictDuplicateCode">
 <property name="min" value="10"/>
</module>
```

清单 5 中的 `min` 属性设置了 CheckStyle 将会标记出的最小重复行数，以供查阅。在这样的情况下，它将只指示出那些至少有 10 行类似或重复的代码块。

图 3 展示了自动构建运行后，清单 5 中的模块设置的结果：

图 **3. CheckStyle** 报告指示代码重复度过高

File C:\dev\brewery-temp\src\com\beer\business\data\BeerDaoImpl.java
Error Description
Found duplicate of 5 lines in C:\dev\brewery-temp\src\com\beer\business\data\BeerDaoImpl.java, starting from line 54
Found duplicate of 5 lines in C:\dev\brewery-temp\src\com\beer\business\data\BeerDaoImpl.java, starting from line 90

重构

在清单 6 中，我用了 清单 4中的重复代码，使用了 *Pull Up Method*重构来降低重复度 —将行为从较大方法提取到一个抽象类方法中：

清单 **6. Pull Up Method**

```
...
} finally {
 closeDbConnection(rs, stmt, conn);
}
...
```

不要忘记编写测试程序

任何时候改变现有代码，您都需要用诸如 JUnit 这样的框架编写相应的自动测试程序。修改现有代码是存在风险的；而将这个风险降到最低的一种方法就是通过测试来验证该行为在现在和将来都有效。

重复代码是难以避免的。我永远不会建议一个团队去努力实现什么 无重复之类的目标，这是不切实际的。然而，确保代码库中的重复代码不会增多这样的目标是可以实现的。使用诸如 PMD 的 CPD 或 CheckStyle 这样的静态分析工具，您能够将整个分析过程作为自动构建的一部分，持续分析，确定代码重复度高的区域。

长方法（大类）

味道：长方法（大类）

度量：源代码行数（SLOC）

工具：PMD、JavaNCSS、CheckStyle

重构: Extract Method、Replace Temp with Query、Introduce Parameter Object、Preserve Whole Object、Replace Method with Method Object

味道

我一直在尝试坚持的一条经验法则是将方法限制在 20 行或 20 行以内。当然，这个原则也可能会有例外，但如果我的方法超过 20 行的话，我就会更仔细地去了解它。通常情况下，长方法和条件复杂度是息息相关的。而大类与长方法之间又有着必然的联系。我可以给您展示一个 2200 行的方法，这个方法是我在需要维护的一个项目上发现的。我将整个含有 25000 行的代码的类打印了出来，让我的同事来找出里面的错误。这么说吧，当我把打印出来的代码沿着走廊卷起来的时候，他们就已经同意我的看法了。

清单 7 中高亮显示的部分展示了一个长方法代码味道示例的一小部分：

清单 **7.** 长方法代码味道

```
public void saveLedgerInformation() {
...
try {
 if (ledger.getId() != null && filename == null) {
   getLedgerService().saveLedger(ledger);
 } else {
   accessFiles().decompressFiles(files, filenames);
 }
 if (!files.get(0).equals(upload)) {
   upload = files.get(0);
   filename = filenames.get(0);
 }
 if (invalidFiles.isUnsupported(filename)) {
   setError(fileName, message.getMessage());
 } else {
 LedgerFile entryFile = accessFiles().add(upload, filename);
 if (fileType != null && FileType.valueOf(fileType) != null) {
   entryFile.setFileType(FileType.valueOf(fileType));
 }
 getFileManagementService().saveLedger(ledger, entryFile);
 if (!FileStatus.OPENED.equals(entryFile.getFileStatus())) {
   getFileManagementService().importLedgerDetails(ledger);
 }
 if (uncompressedFiles.size() > 1) {
   Helper.saveMessage(getText("ledger.file"));
 }
 if (user.getLastName() != null) {
   SearchInfo searchInfo = ServiceLocator.getSearchInfo();
   searchInfo.setLedgerInfo(null);
   isValid = false;
   setDefaultValues();
   resetSearchInfo();
   if (searchInfoValid && ledger != null) {
     isValid = true;
   }
 }
} catch (InvalidDataFileException e) {
 ResultType result = e.getResultType();
 for (ValidationMessage message : result.getMessages()) {
   setError(fileName, message.getMessage());
 }
 ledger.setEntryFile(null);

} ...
```

度量

在过去的几年里，SLOC 度量方法被误认为是高效率的象征。尽管我们都知道，并不一定是行数越多越好。但说到复杂度，SLOC 可是一个有用的度量方法。一个方法（或类）的行数越多，将来维护其代码就可能越难。

工具

清单 8 中的脚本为长方法（大类）找到了 SLOC 度量方法：

清单 **8.** 识别过大的类和方法的 **Gant** 脚本

```
target(findLongMethods:"runs static code analysis"){
Ant.mkdir(dir:"target/reports")
Ant.taskdef(name:"pmd", classname:"net.sourceforge.pmd.ant.PMDTask",
  classpathref:"build.classpath")
Ant.pmd(shortFilenames:"true"){
  codeSizeRules.each{ rfile ->
   ruleset(rfile)
  }
  formatter(type:"xml", tofile:"target/reports/pmd_report.xml")
  formatter(type:"html", tofile:"target/reports/pmd_report.html")
  fileset(dir:"src"){
    include(name:"**/*.java")
  }
 }
}
```

我又使用了 Gant 访问 Ant API 来执行 Ant 任务。在清单 8 中，我调用 PMD 静态分析工具来搜索代码库中的长方法。PMD（连同 JavaNCSS 与 CheckStyle）也可以用于查找长方法、大类以及其他代码味道。

重构

清单 9 展示了用 *Extract Method* 重构来减少 清单 7 中的长方法代码味道的一个例子。将清单 7 的方法中的行为提取到清单 9 的代码中以后，我就可以从清单 7 的 `saveLedgerInformation()` 方法中调用新建的 `isUserValid()` 方法了：

清单 9. Extract Method 重构

```
private boolean isUserValid(User user) {
 boolean isValid = false;
 if (user.getLastName() != null) {
   SearchInfo searchInfo = ServiceLocator.getSearchInfo();
   searchInfo.setLedgerInfo(null);
   setDefaultValues();
   resetSearchInfo();
   if (searchInfoValid && ledger != null) {
     isValid = true;
   }
 }
 return isValid;
}
```

通常，长方法和大类也暗示着存在其他代码味道，如条件复杂度和重复代码。因此，找到这些长方法和大类也就可以修复其他的问题了。

太多导入

味道：太多导入

度量：传出耦合（每个类的扇出（fan-out））

工具：CheckStyle

重构：Move Method、Extract Class

味道

太多导入表明一个类过多地依赖于其他的类。您会注意到，由于一个类与很多其他的类耦合得太紧密，修改这个类会导致必须对很多其他的类进行修改，这时就说明这个类存在这种代码味道了。清单 10 中的多个导入就是一个例子：

清单 **10.** 一个类中的多个导入

```
import com.integratebutton.search.SiteQuery;
import com.integratebutton.search.OptionsQuery;
import com.integratebutton.search.UserQuery;
import com.integratebutton.search.VisitsQuery;
import com.integratebutton.search.SiteQuery;
import com.integratebutton.search.DateQuery;
import com.integratebutton.search.EvaluationQuery;
import com.integratebutton.search.RangeQuery
import com.integratebutton.search.BuildingQuery;
import com.integratebutton.search.IPQuery;
import com.integratebutton.search.SiteDTO;
import com.integratebutton.search.UrlParams;
import com.integratebutton.search.SiteUtil;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collection;
import java.util.Collections;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.apache.log4j.Logger;
...
```

度量

找到带有太多责任的类的一个方法就是通过 传出耦合度量方法，亦指 扇出复杂度。扇出复杂度给被分析的类所依附的每一个类赋值 1。

工具

清单 11 展示了一个用 CheckStyle 设置最大扇出复杂度数的例子：

清单 **11.** 使用 **CheckStyle** 设置最大扇出复杂度

```
<module name="ClassFanOutComplexity">
 <property name="max" value="10"/>
</module>
```

Refactoring to Patterns

工厂方法模式是应用重构时可以实现的多种设计模式之一。用工厂方法创建类无需显式定义正在创建的实际的类。这个模式可以使界面比实现类更简单。当根据代码味道实现重构时，您也可以使用其他的设计模式；参见 参考资料查看专门研究这个概念的书籍的链接。

重构

修复由于太多导入而引发的耦合过紧的方法有很多种。对于诸如 清单 10中那样的代码来说，可用的重构就包括 *Move Method*重构：将方法从单独的 * `Query` 类移动到 Java 界面，并定义所有 `Query` 类必须实现的通用方法。然后再使用 工厂方法模式，这样耦合度就与界面相关联了。

通过使用 Gant 自动构建脚本执行 CheckStyle Ant 任务，我可以搜索代码库，查找过多依赖于其他类的类。当修改这些类中的代码时，就能够实现特定的重构（比如 Move Method）和特定的设计模式，以逐步改进可维护性。

重构……要尽早且要经常进行

持续集成（Continuous Integration，CI）就是经常集成变更。正如其典型的实现方式一样，每当对项目的版本控制储存库做出一个更改时，运行于独立机器上的自动 CI 服务器就会触发一个自动构建。为了确保 清单 3和 清单 8中的脚本可以在对数据库做出更改时一致地运行，您需要配置一个诸如 Hudsona 这样的 CI 服务器（参见 参考资料）。Hudson 是以 WAR 文件的形式发布的，您可以将它放入任何 Java Web 容器中。

由于 清单 3和 清单 8中的例子使用了 Gant，下面我就简要介绍一下配置 Hudson CI 服务器以运行 Gant 脚本的步骤：

1. 在 Hudson 的仪表板上为 Hudson 安装 Gant 插件：首先选择 **Manage Hudson**，再选择 **Manage Plugins**，然后选择 **Available**选项卡。在这个选项卡上选中 Gant 插件复选框，然后单击 **Install**按钮。
2. 重新启动 Web 容器（例如，Tomcat）。
3. 选择 **Manage Hudson**，然后选择 **Configure System**。在 **Gant installation**部分，键入一个惟一的名称以及 Groovy 安装到运行 Hudson 的机器上的位置。保存所作更改。
4. 返回到仪表板（选择 **Hudson**链接），选择一个 **existing Hudson Job**，再选择 **Configure**，然后单击 **Add build step**按钮，选择 **Invoke Gant script**选项。

配置 Hudson，使其运行使用 Gant 编写的自动构建脚本。一旦诸如长方法和条件复杂度这样的代码味道被引入到代码库中，您立刻就会得到与它们相关的度量方法的反馈。

其他味道与重构

并非所有的味道都有相关的度量方法。但是，静态分析工具能够揭露的味道不止我所示范的这些。表 1 列举了其他的代码味道、工具、以及可能的重构例子：

表 **1.** 其他味道与重构

味道	工具	重构
死代码	PMD	Remove Code
临时字段	PMD	Inline Temp
不一致 / 拘谨（uncommunicative）的名称	CheckStyle、PMD	Rename Method、Rename Field
长参数列表	PMD	Replace Parameter with Method、Preserve Whole Object、Introduce Parameter Object

本文提供了一种使代码味道与一种度量方法相关的模式，这种度量方法可以配置为通过自动静态分析工具标记。您可以使用或不使用特定的设计模式来进行重构。这为您提供了一个以可重复的方式一致地查找和修复代码味道的框架。我坚信本文的例子也有助于您使用静态分析工具来查找本文未涉及到代码味道。

让开发自动化: 文档化一键通

自动生成开发人员和用户文档

项目的文档化通常都是交付软件产品时不可避免的难题。但是想象一下如果仅需单击一个按钮就能够生成文档呢？在本期的 让开发自动化 中，自动化专家 Paul Duvall 阐述了如何运用开源工具自动生成统一建模语言（Unified Modeling Language，UML）图、构建图、实体关系图（entity-relationship diagram，ERD）乃至用户文档。

很少有软件开发人员愿意为他们的软件开发项目编写文档。然而，除非您永远不想完成您的项目，或者您永远独立进行软件开发，又或者您没有用户 —这对一个项目来说可不是一个好现象，否则您就需要用一种方式来向别人交待您的软件的用途。有些开发者误解了 Agile Manifesto 的“可运行的软件 优于全面的文档”的说法，以为根本不需要 任何的文档资料（请参阅 参考资料）。另一方面，多余的文档对用户或其他开发人员来说也是一个负担。我通常寻找一种中间办法。您猜对了：本文将向您展示如何运用 自动化来简化项目文档的生成过程，从而减轻这方面的负担。

关于本系列

作为开发人员，我们致力于为用户自动化流程；但许多开发人员疏忽了自动化我们自己的开发流程的机会。为此，我们编写了 让开发自动化系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍 何时以及 如何成功应用自动化。

按照我的经验，有两个关键性的问题制约着软件开发的文档化。第一个问题，似乎没有人会去阅读文档。第二个常见问题就是几乎是在编写文档的同时，它就已经过时了。这两个问题是有着因果联系的：如果文档总是保持最新，那么人们就更有可能去阅读它。自动化生成文档便可以解决这两个问题，它可以保持文档的时效性，从而使它对您的软件用户更加有用。

自动化也可能对其他类型的文档有利，但在本文中我将着重阐述如何让那些令人痛苦的文档化任务自动化（请参阅 参考资料查找下面列表中提到的工具的链接）：

- 使用 UMLGraph 生成当前源代码的 **UML**图。
- 使用 SchemaSpy 创建 实体关系图（**ERD**），归档数据库中的表格和关系。
- 使用 Grand 生成构建目标以及它们之间的关系的 Ant 构建图。
- 使用 Doxygen 生成 源代码文档。
- 使用 DocBook 制作 用户文档。

我将使用一般的方法阐述以下内容：

1. 描述手动执行每一项任务遇到的问题。
2. 呈现一个结合使用 Apache Ant 和相关的文档 / 图表生成工具实现自动化的例子。

3. 展示一个基于代码示例的脚本生成文档的图象。

和本系列前面的文章一样，文中所有示例都使用可免费获得的开源工具，您可以在自己的项目中使用。有些工具（例如，UMLGraph 和 Grand）会使用一个附带的 GraphViz 工具，该工具要用到一个由特定工具生成的 .dot 文件。

将代码反向工程到 UML 中

我曾遇到过一些拥有十分美观的 UML 图的项目 —在项目的初始阶段。问题是，在某种特殊情况下，技术负责人无法使模型与源代码同步。或者需要将宝贵的时间浪费在将源代码 手动反向工程到模型中。这两种情况都不尽如人意。如果模型无法如实地展示签入到版本控制库中的代码，那么这个 UML 图构造的再美观也毫无意义。如果您没有根据实际的代码制定决策，那么您可能就要面临随之而来的很多问题。

您可以在构建过程中生成图示，并建立一个持续集成（Continuous Integration，CI）环境来即时地（或定期地）创建图示。这样您就能够制作出有益于决策、易于创建且始终反映最新情况的图示了。

清单 1 使用 Ant、UMLGraph 以及 Graphviz 对源代码进行文档化：

清单 **1.** 使用 **UMLGraph** 文档化工具的 **Ant** 脚本

```
<property name="reports.dir" value="${basedir}/reports"/>
 <mkdir dir="${reports.dir}"/>
 <path id="project.path">
   <pathelement path="${basedir}/src"/>
   <pathelement path="${basedir}/lib"/>
 </path>
 <javadoc sourcepath="${basedir}/src" destdir="${reports.dir}"
   classpathref="project.path" access="private">
   <doclet name="org.umlgraph.doclet.UmlGraphDoc"
     path="UMLGraph.jar">
       <param name="-attributes" />
       <param name="-enumerations" />
       <param name="-enumconstants" />
       <param name="-operations" />
       <param name="-qualify" />
       <param name="-types" />
       <param name="-visibility"/>
   </doclet>
 </javadoc>
 <apply executable="dot" dest="${reports.dir}" parallel="false">
   <arg value="-Tpng"/>
   <arg value="-o"/>
    <targetfile/>
    <srcfile/>
    <fileset dir="${reports.dir}" includes="*.dot"/>
    <mapper type="glob" from="*.dot" to="*.png"/>
 </apply>
```

清单 1 中，我结合使用 UMLGraph 和 Javadoc 在 Javadoc HTML 报告内部生成一些基本的 UML 类图。为了自定义展示在每一个类图中的信息，调用 `UmlGraphDoc` 时，我传递了如下属性：

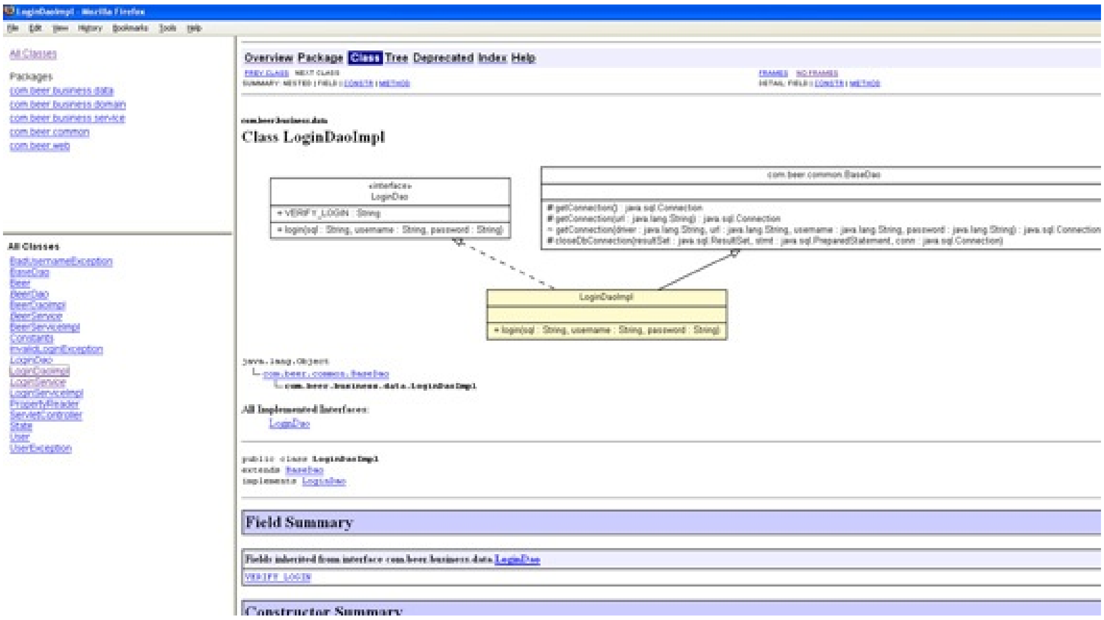
使用 GraphViz 制图

要正常使用 UMLGraph，您必须先安装 Graphviz 工具（请参阅 参考资料，并且计算机的系统路径中必须包含 Graphviz .dot 文件。

- `-attribute` 为类显示字段。
- `-enumeration` 展示不同的枚举。
- `-enumconstant` 为枚举展示可能的值。
- `-operation` 为类显示 Java 方法。
- `-qualify` 显示完全限定类名。
- `-type` 显示参数数据类型和返回类型
- `-visibility` 展示字段和方法修改语：`public` 、`protected` 、`private` 或 `default` 。

图 1 展示了用 UMLGraphshows 生成到 HTML 中的 `LoginDaoImpl` 类及其关系的 UML 图例：

图 **1.** 用 **UMLGraph** 生成的 **UML** 图



请 单击此处查看完整图形。

UMLGraph 能够生成更复杂的关系和其他细节。但是，即便是这个简单例子中极其基本的 UML 类图，就已经提供了相当多的信息。它也可以根据当前的代码库快捷、形象地展示软件。它防止了“代码 应该是这样的”这种陈述方式，并且有益于做出更好的决策（关于描述了多种可以自定义 UMLGraph 输出的属性的链接，请参阅 参考资料）。

数据库文档化

自动创建最新的 UML 图将使您获益良多，同样，自动地可视化表示数据库也可以让您受益匪浅。实体关系图（ERD）是实现数据库可视化的最流行的图示类型。大多数创建 ERD 的工具（例如，ERWinfor）都需要手动生成 ERD。虽然我将要示范的工具 SchemaSpy 无法与现存的一些更复杂的工具媲美，但是它能够提供数据库的高级 ERD 视图 —以及约束、关系等。而且，通过自动构建来运行它，您就可以轻松地从您的版本控制库中检查数据定义语言（Data Definition Language，DDL）的最新显示。

清单 2 中的 Ant 脚本使用 SchemaSpy 工具来创建 Javadoc 格式的文件：

清单 **2.** 结合使用 **SchemaSpy**、**Ant** 和 **Javadoc**

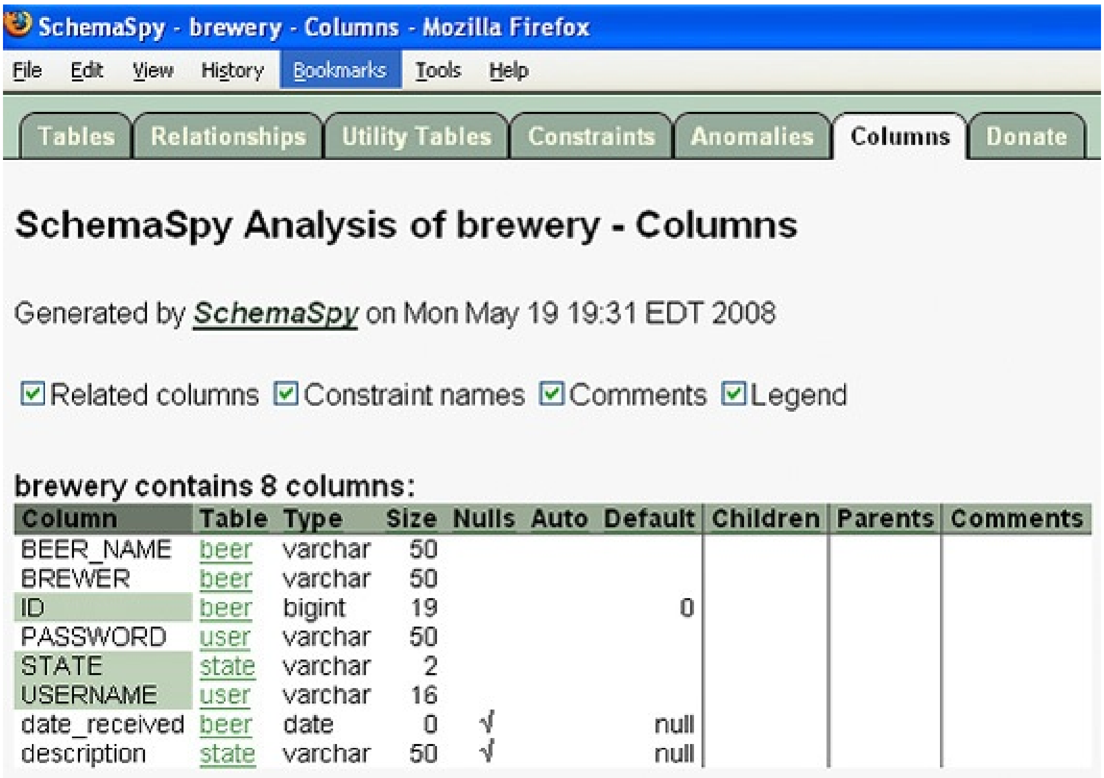
```
<property name="reports.dir" value="${basedir}"/>
<java jar="schemaSpy_3.1.1.jar"
 output="${reports.dir}/output.log"
 error="${reports.dir}/error.log"
 fork="true">
 <arg line="-t mysql"/>
 <arg line="-host localhost"/>
 <arg line="-port 3306"/>
 <arg line="-db brewery"/>
 <arg line="-u root"/>
 <arg line="-p sa"/>
 <arg line="-cp mysql-connector-java-5.0.5-bin.jar"/>
 <arg line="-o ${reports.dir}"/>
</java>
```

清单 2 使用 `java` Ant 任务调用 SchemaSpy，传递了很多属性：

- **`-t`** 为数据库类型（有效值为 `mysql`、`ora`、`db2`，等等。)
- **`-host`** 为托管数据库的计算机名。
- **`-port`** 为数据库 URL 的端口数。
- **`-u`** 为数据库用户名。
- **`-p`** 为数据库密码。
- **`-cp`** 为类路径（用于指示数据库驱动程序 JAR 文件的位置）。
- **`-o`** 为输出目录的位置。

这些 SchemaSpy 的命令行属性用于生成显示 ERD 的 HTML 文件，如图 2 所示：

图 **2.** 用 **SchemaSpy** 和 **Ant** 创建的 **ERD**



通过结合使用多种工具、作为构建的一部分针对数据库执行 ERD 生成脚本，并调度 ERD 生成，您就可以在开发过程中轻松、快速地做出很多数据库决策。

图解构建过程

通常，构建脚本在相关的目标之间有着复杂的关系。我曾见过许多构建脚本足有 1000 多行，含有大量多余的重复行。如果没有自动构建的高级视图，这些构建脚本会很难处理。要想快速确定一个构建过程正在处理的内容，一个有效的方法就是创建一个目标和关系图。Ant 目标的一个可视模型能够帮助您作出更有效、更明智的决策，以改进构建脚本的设计。

结合使用 Ant、Grand 和 Graphviz，您就可以创建出构建目标的精确的可视表示。有很多的工具都旨在可视化表示 Ant 目标。而 Grand 的一个优势便在于它运用了 Ant API，这样无论 Ant 脚本是否完全有效，它都可以创建图示。

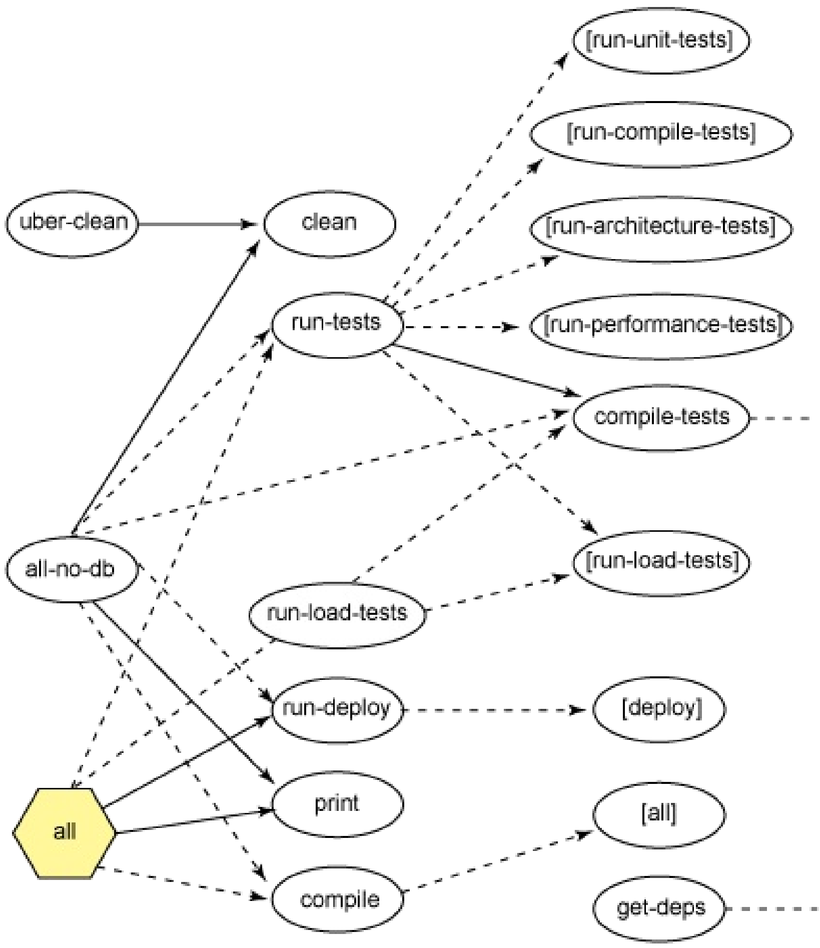
清单 3 中的 Ant 脚本展示了如何运用 Grand 工具来创建 Ant 脚本目标的可视表示，包括目标间的依赖性：

清单 **3.** 使用 **Ant** 和 **Grand** 创建 **Ant** 目标的图示

```
<target name="create-ant-diagram">
 <property name="file.type" value="pdf"/>
 <typedef resource="net/ggtools/grand/antlib.xml"
   classpath="grand-1.8.jar"/>
 <grand output="build.dot" buildfile="${basedir}/build.xml"/>
   <exec executable="dot">
     <arg line="-T${file.type} -Gsize=11.69,8.27 -Grotate=90 -o build.${file.type}
   ${grand.output.file}"/>
   </exec>
</target>
```

在清单 3 中，首先，我通过指向 `grand-1.8.jar` 创建了任务。接着，我在 `buildfile` 属性中为正在分析的构建文件命名。`output` 属性是我正在创建的 Graphviz 文件（我将其命名为 `build.dot`）。最后，我调用了 `dot` 可执行文件生成 `buildfile` 的构建目标的 PDF。图 3 展示了这种文件的一个例子：

图 **3.** 使用 **Ant**、**Grand** 和 **GraphViz** 生成 **Ant** 目标的图示



我可以运用这些图表分析已完成的项目，或者为开发中的项目优化构建过程。

代码文档化

如果我想记住为什么创建了一个新的 Java 类或方法，那么我会写一个 Java 代码注释。但如果我想更全面地了解新建的类、方法和属性，那么我就需要一个代码文档化的工具了。多年以来，Java 平台都将 Javadoc 作为生成代码文档的方法，用于代码库中的所有类。事实上，`javadoc` 任务已成了 Ant 中的标准，它也确实能够生成代码文档。然而，它需要由开发人员负责将一些难看的 HTML 标记嵌入到 Java 源代码注释中。例如，清单 4 所示的运用 Javadoc 的代码注释：

清单 **4. Javadoc** 的代码注释

```
/**
 This is <i>the</i> LoginServiceImpl  class.
 <p>
 Refer to <a href="url.html">
 LoginService overview</a> for more details.
 <p>
 There is one type of supported {@link LoginServiceImpl }:
 <ul>
   <li>{@link LoginServiceImpl } (this class)</li>
 </ul>
*/
```

如果使用 Doxygen，代码注释就会更简洁，省去了嵌入的 HTML 标记，如清单 5 所示：

清单 **5.** 使用 **Doxygen** 文档化工具的 **Ant** 脚本

```
/**
This is <i>the</i> LoginServiceImpl class.
Refer to \ref LoginService for more details.
There is one type of supported LoginService:
- LoginServiceImpl (this class)
*/
```

Javadoc 和 Doxygen 都支持自动生成代码文档，但是使用 Doxygen 比较简单，它提供了一种更直观的方法来用注释标注源代码。清单 6 中的 Ant 代码片断展示了 Doxygen 最基本的使用方法：

清单 **6.** 使用 **Doxygen** 文档化工具的 **Ant** 脚本

```
<target name="generate-doxygen">
 <taskdef name="doxygen" classname="org.doxygen.tools.DoxygenTask"
   classpath="ant-doxygen.jar" />
 <doxygen configFilename="Doxyfile">
 <property name="INPUT" value="${src.dir}" />
 <property name="RECURSIVE" value="yes" />
 </doxygen>
</target>
```

在清单 6 中，`Doxyfile` 的 `configFilename` 值是由 Doxygen 自动生成的配置文件的名称；您可以重新定义。图 4 展示了一个 `LoginDaoImpl` 类的 HTML 报告的例子，它是由清单 6 中的 Doxygen 示例生成的：

图 **4. HTML Doxygen** 文档



请 单击此处查看完整图形。

通过在自动构建中创建 Doxygen 文档，您可以获得有关某个 API 的最新的使用信息。

用户文档

到目前为止，您可能一直在想，自动生成技术文档固然不错，但是最头疼的是创建 用户文档。在编写自己的用户文档时，我发现大量的时间都浪费在文档内部的复制、粘贴上。分析了该过程之后，我决定将诸如版本号、文件名等内容定义到单一源中，以此来实现这些内容的生成自动化。而且我可以运用 XML 和可扩展样式语言（Extensible Stylesheet Language，XSL）将内容从格式化中分离出来，这样我只要对一个样式做一个简单的修改就可以使文档的观感发生显著的变化。

DocBook 工具已存在数年之久，它使您能够用 XML 定义文档，并能够以包括 HTML 和 PDF 在内的多种格式生成文档。DocBook 提供了一种为编写者定义模板的模式。

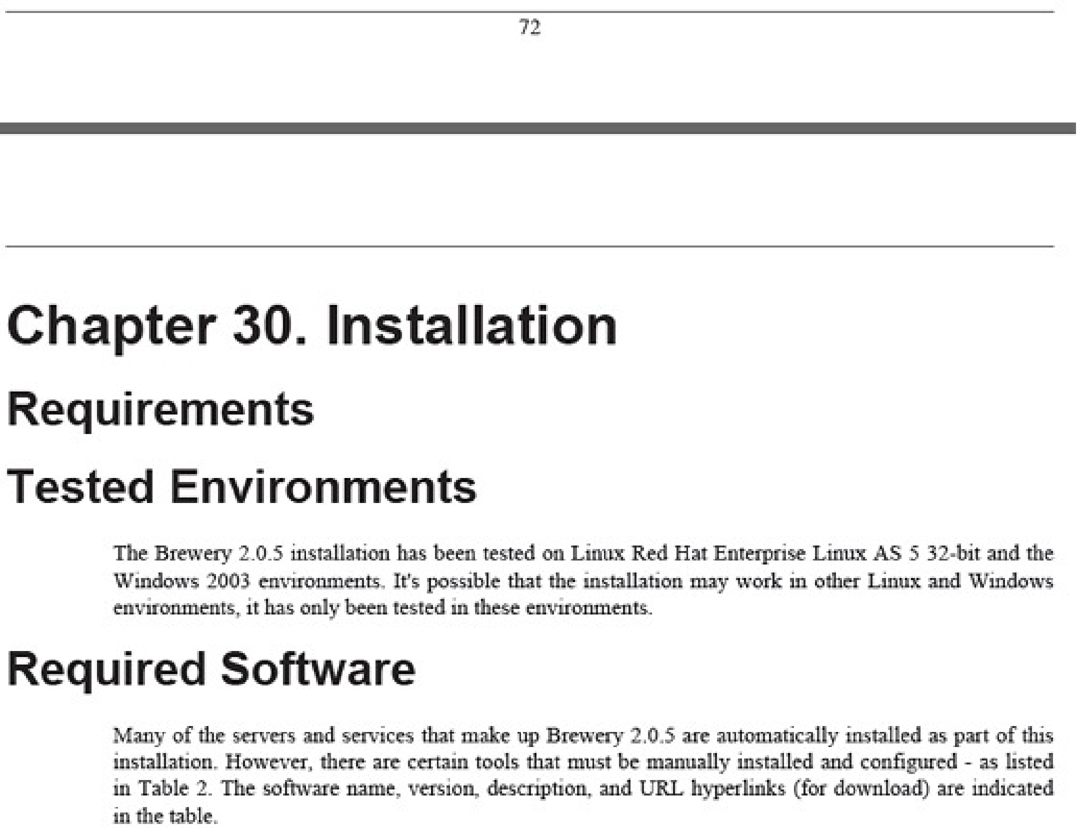
清单 7 结合使用 Ant、一个 DocBook XSL 文件和 XSL Formatting Objects (XSL-FO) 来生成 PDF 格式的用户文档（请参阅 参考资料）：

清单 **7.** 使用 **Ant**、**DocBook** 和 **FO** 创建 **PDF**

```
<target name="pdf" depends="init" description="Generates PDF files from DocBook XML">
 <property name="fo.stylesheet" value="${docbook.xsl.dir}/fo/docbook.xsl" />
 <xslt style="${fo.stylesheet}" extension=".fo" basedir="${src.dir}" destdir="${fo.dir}"
   <classpath refid="xalan.classpath" />
   <include name="${guide}.xml" />
 </xslt>
 <property name="fop.home" value="lib/fop-0.94" />
 <taskdef name="fop" classname="org.apache.fop.tools.anttasks.Fop">
   <classpath>
     <fileset dir="${fop.home}/lib">
       <include name="*.jar" />
     </fileset>
     <fileset dir="${fop.home}/build">
       <include name="fop.jar" />
       <include name="fop-hyph.jar" />
     </fileset>
   </classpath>
 </taskdef>
 <fop format="application/pdf" fofile="${fo.dir}/${guide}.fo"
   outfile="${doc.dir}/${guide}.pdf" />
</target>
```

图 5 展示了由清单 7 中的脚本生成的 PDF 的一部分：

图 **5.** 用户指南文档的 **PDF**


72

Chapter 30. Installation

Requirements

Tested Environments

The Brewery 2.0.5 installation has been tested on Linux Red Hat Enterprise Linux AS 5 32-bit and the Windows 2003 environments. It's possible that the installation may work in other Linux and Windows environments, it has only been tested in these environments.

Required Software

Many of the servers and services that make up Brewery 2.0.5 are automatically installed as part of this installation. However, there are certain tools that must be manually installed and configured - as listed in Table 2. The software name, version, description, and URL hyperlinks (for download) are indicated in the table.

由于我的全部内容都是用基于文本的 XML 定义的，所以我可以轻松地写出令文档发生显著变化的脚本。此外，我可以将这些文档做为构建过程的一部分生成，进而为定义诸如版本、定义、缩写、文件名以及目录之类的元素提供一个单一的标准源。

文档化无需手动操作

希望阅读本文之后，您能够了解到可以通过有效的方法自动生成文档。在我看来，文档化最关键的地方就在于交流，而非生成文档的过程。我向您描述的文档化工具和技术能够帮助您保持文档的精确性和实效性，从而清除了文档之所以被忽视的症结所在 —信息过时。

让开发自动化: 利用 Ivy 管理依赖项

使用公共存储库和 *Apache Ant* 共享其他项目的源代码

管理项目和工具之间的源代码依赖项往往非常困难，但并不一定总是如此。在这一期“ 让开发自动化”中，自动化专家 Parl Duvall 介绍了如何利用 Apache Ant 项目中的 Ivy 依赖项管理器来处理所有重要 Java 项目必须管理的无数依赖项。

实际上，所有软件开发项目都必须依靠来自其他项目的源代码。例如，许多项目可能依靠 log4j 等日志记录工具和 Struts 之类的 Web 框架。您的开发团队不会维护其他项目的源代码，但要依靠其 API 来实现项目中的定制软件。您的软件所依靠的其他项目数量越多（包括这些项目自身的依赖项），构建软件就变得越复杂。

关于本系列

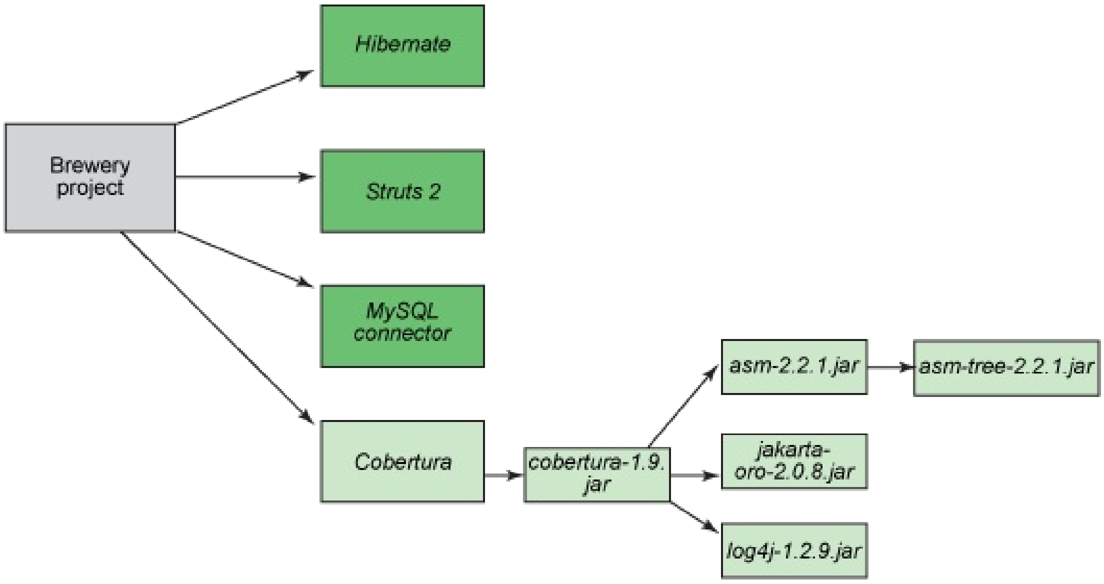
作为开发人员，我们致力于为用户自动化流程；但许多开发人员疏忽了自动化我们自己的开发流程的机会。为此，我们编写了 让开发自动化系列文章，专门探讨软件开发流程自动化的实践应用，为您介绍 何时以及 如何成功应用自动化。

我已经看到许多团队使用各种不完善的技术，尝试解决这种难题：

- 将全部有依赖关系的项目（JAR 文件）放在一个目录中，此目录将签入项目的版本控制存储库。这种技术不必要地增加了存储库的大小，使得管理版本差异极为困难。
- 将有依赖关系的 JAR 分配到一个公共文件服务器上，使团队无法控制版本更改。
- 手动将 JAR 文件复制到各开发人员工作站上的指定位置。这种方法使得确定丢失的文件或修正版本极为困难。
- 执行一条 HTTP `Get` 命令，将文件下载到开发人员的工作站，手动执行或将其作为自动构建的一部分。这种技术会造成未受管理的 JAR 文件。

我参加过一个中型项目，包含 1,000 个 Java 类和 100 多个有依赖关系的 JAR 文件。（我们选择了第一种不完美的技术：将所有 JAR 签入项目的版本控制存储库。）图 1 显示了可能在此类项目中看到的一小部分依赖项的类型：

图 **1. Web** 开发项目中的 **JAR** 依赖项示例



Transfixed on transitive dependencies

传递依赖（*Transitive dependency*）是一个复杂的术语，但表示的是 Ivy 提供的一种简单而强大的特性。某些 JAR 文件依赖于其他 JAR，这样才能正常工作。使用 Ivy，您只需一次性声明一个组件的依赖项。此后，仅需了解一个项目的主要 JAR 文件，而无需了解它的所有 JAR 文件依赖项。如果您体验过手动查找依赖项的痛苦——无论是通过文档还是通过研究代码，您就会发现，此特性本身就值得您付出时间在项目中配置 Ivy。参见本文 依赖于依赖项一节了解更多细节。

图 1 表现出，Brewery 项目的源代码依赖于 Hibernate、Struts 2、MySQL Connector 和 Cobertura。而 Cobertura 又依赖其他 JAR，如 asm-2.2.1.jar、jakarta-oro-2.0.8.jar 和 log4j-1.2.9.jar。此外，asm-2.2.1.jar 依赖 asm-tree-2.2.1.jar。这仅仅是可能出现的各类嵌套依赖项的一个简单示例。即便是某个 JAR 的版本不正确，您也会体验到难以排除的问题，例如编译错误或意料之外的行为。

Apache Maven 构建管理和项目管理工具已经吸引了 Java 开发人员的注意。Maven 引入了 JAR 文件公共存储库的概念，可通过公开的 Web 服务器访问（称为 ibiblio）。Maven 的方法减少了 JAR 文件膨胀的情况，不会占用大多数版本控制存储库。但使用 Maven 时，它会鼓励您采用其 “惯例优于配置” 的方法来构建软件，这会制约您定制构建脚本的灵活性。

如果您多年来一直使用 Apache Ant，现在希望获得使用公共存储库的优势，又该如呢？您是否不得不接受 Maven 的构建方法来获得这些收益？幸运的是，答案是否定的，这是由于一种称为 Apache Ivy 的工具 —Ant 的一个子项目。Ivy 提供了最一致、可重复、易于维护的方法，来管理项目的所有构建依赖项（在 参考资料部分中可以找到 Maven 和 Ivy 的比较）。这篇文章介绍了安装和配置 Ivy 来管理依赖项的基础知识，指出了可参考的更多信息。

入门

开始使用 Ivy 非常简单，只需创建两个 Ivy 特有的文件，添加一些 Ant 目标即可。Ivy 特有的文件是 ivy.xml 和一个 Ivy 设置文件。ivy.xml 文件中列举了项目的所有依赖项。ivysettings.xml 文件（可以随意为此文件命名）用于配置从中下载有依赖关系的 JAR 文件的存储库。

清单 1 展示了一个简单的 Ant 脚本，它调用了两个 Ivy 任务：`ivy:settings` 和 `ivy:retrieve` 。

清单 **1.** 使用 **Ivy** 的简单 **Ant** 脚本

```
<target name="init-ivy" depends="download-ivy">
 <ivy:settingsfile="${basedir}/ivysettings.xml" />
 <ivy:retrieve/>
</target>
```

在 清单 1中，`ivy:settings` 定义了 Ivy 设置文件。对 `ivy:retrieve` 的调用从 ivy.xml 声明的一个存储库中检索 JAR 文件。

安装 Ivy

下载并使用 Ivy 的方法有几种。第一种是手动将 Ivy JAR 文件下载到 Ant lib 目录中，也可下载到 Ant 脚本的类路径中定义的某个目录中。我迷上了自动化，所以更倾向于使用自动化替代方案：下载 Ivy 的 JAR 文件，在 Ant 目标中配置类路径。清单 2 展示了这种技术的示例：

清单 **2.** 使用 **Ant** 自动安装 **Ivy**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<project name="test-ivy" default="init-ivy" basedir="."
  xmlns:ivy="antlib:org.apache.ivy.ant" xmlns="antlib:org.apache.tools.ant">
 <property name="ivy.install.version" value="2.0.0-beta2" />
 <property name="ivy.home" value="${user.home}/.ant"/>
 <property name="ivy.jar.dir" value="${ivy.home}/lib" />
 <property name="ivy.jar.file" value="${ivy.jar.dir}/ivy.jar" />

 <taskdefresource="org/apache/ivy/ant/antlib.xml"
    uri="antlib:org.apache.ivy.ant" classpath="${ivy.jar.dir}/ivy.jar"/>

 <target name="download-ivy">
   <mkdir dir="${ivy.jar.dir}"/>
   <get src="http://www.integratebutton.com/repo/
      ${ivy.install.version}/ivy-2.0.0-beta2.jar"
     dest="${ivy.jar.file}"usetimestamp="true"/>
 </target>

</project>
```

清单 2中的第二行定义了 XML 名称空间。`antlib` 在 ivy.jar 文件中引用 antlib.xml。其余的 `xmlns` 指明了 `ivy` Ant 任务的完全限定路径。`${user.home}/.ant` 的 `ivy.home` 值是 ivy.jar 文件下载的目标位置。`taskdef` 定义了 `ivy` Ant 任务，引用其类路径的位置。`download-ivy` 目标下载 ivy-2.0.0-beta2.jar 并使用 `dest` 属性为其重命名。

一旦下载并配置了 Ivy，就可以使用任意 Ivy Ant 任务（如 清单 1中调用的两个任务）。

创建配置脚本

ivy.xml 文件是必不可少的，您在此文件中定义项目的全部有依赖关系的 JAR。清单 3 展示了一个示例：

清单 **3.** 在 **ivy.xml** 中定义依赖项

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="./config/ivy/ivy-doc.xsl"?>
<ivy-module version="1.0">
 <info organisation="com"module="integratebutton" />
 <dependencies>
   <dependency name="hsqldb" rev="1.8.0.7" />
   <dependency name="pmd" rev="2.0" />
   <dependency name="cobertura" rev="1.9"/>
   <dependency name="checkstyle" rev="4.1" />
   <dependency name="junitperf" rev="1.9.1" />
   <dependency name="junit" rev="3.8.1" />
 </dependencies>
</ivy-module>
```

请注意，清单 3未表示任何文件位置或 URL，允许您转到其他目录位置，而无需更改依赖项列表。 `info` 元素中的 `organisation` 属性标识了组织类型（如 .net、.org 或 .com）。后接 `module` 名称。此模块的依赖项列表遵循一种命名规范，在下一个清单中您将更清晰地看出此规范。目前，只需记住 `dependency name="cobertura" rev="1.9"` 将转换为 cobertura-1.9.jar 即可。

清单 4 是 Ivy 设置文件的示例。它定义了 清单 3中 ivy.xml 文件所用的存储库位置和相关模式。

清单 **4. Ivy** 设置文件

```
<ivysettings>
 <settings defaultResolver="chained"/>
 <resolvers>
   <chain name="chained" returnFirst="true">
     <filesystem name="libraries">
       <artifact pattern="${ivy.conf.dir}/repository/[artifact]-[revision].[type]" />
     </filesystem>
     <url name="integratebutton">
       <artifact pattern="http://www.integratebutton.com/repo/[organisation]/[module]/
          [revision]/[artifact]-[revision].[ext]" />
     </url>
     <ibiblio name="ibiblio" />
     <url name="ibiblio-mirror">
       <artifact pattern="http://mirrors.ibiblio.org/pub/mirrors/maven2/[organisation]/
          [module]/[branch]/[revision]/[branch]-[revision].[ext]" />
     </url>
   </chain>
 </resolvers>
</ivysettings>
```

清单 4中的 `filesystem` 元素定义了本地工作站上的位置模式。两个 `url` 元素定义了可用于下载 JAR 文件的多个位置：第一个元素定义了受我控制的 integratebutton.com 上的一个自定义存储库；第二个元素定义了包含大量开源 JAR 文件的外部 Maven 存储库（不受我控制）。如果 Ivy 无法从第一个存储库下载 —比如此存储库宕机，或者文件未在指定位置 —它将尝试第二个位置。优点在于，一旦 Ivy 下载了一个 JAR，它就会将文件置入您的本地文件系统，不必再为每一次构建重新下载这些文件。

依赖于依赖项

一个模块常常要依赖其他模块。例如，在 图 1中可以看到，cobertura-1.9.jar 文件的多个依赖项中包括 asm-2.2.1.jar，而 asm-2.2.1.jar 又依赖于 asm-tree-2.2.1.jar。如果没有像 Ivy 这样的工具，您就需要确保类路径中存在这些 JAR 的正确版本，保证 JAR 版本之间不存在冲突。而使用 Ivy，您只需定义 `cobertura` 模块及其所有依赖模块，如清单 5 中所示的 ivy.xml 文件那样。切记，这个 ivy.xml 文件与 cobertura-1.9.jar 文件位于同一目录。

清单 **5.** 在 **ivy.xml** 文件中定义依赖项

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">
 <info organisation="cobertura" module="cobertura"revision="1.9"/>
 <configurations>
   <conf name="master"/>
 </configurations>

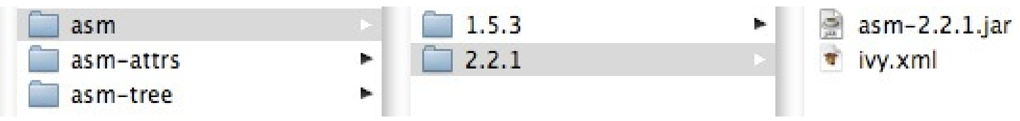
 <publications>
   <artifact name="cobertura" type="jar" conf="master" />
 </publications>

 <dependencies>
   <dependency org="objectweb" name="asm" rev="2.2.1" conf="master"/>
   <dependency org="jakarta" name="oro" rev="2.0.8" conf="master"/>
   <dependency org="apache" name="log4j" rev="1.2.9" conf="master"/>
 </dependencies>
</ivy-module>
```

清单 5中特别强调的依赖项定义了 `objectweb org` 和名称 `asm` 以及要使用的特定修订版。Ivy 将此信息与 ivysettings.xml 文件中的存储库定义（如 清单 4所示）一起使用，用于下载 JAR 文件的依赖项。

图 2 展示了符合 清单 4所示 ivysettings.xml 文件配置的存储库中的目录结构：

图 **2.** `asm` 模块的目录结构



请注意，图 2 展示了一个 ivy.xml 文件（见清单 6），它定义了 `asm` 的依赖项。 在清单 6 中，针对 `asm` 模块的 ivy.xml 文件片段表示了它惟一的依赖项 —asm-tree-2.2.1.jar：

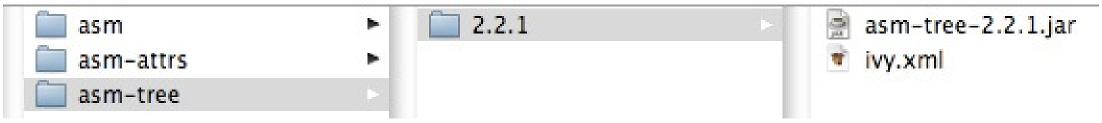
清单 6. 为 `asm` 定义依赖项的 ivy.xml

```
...
<dependencies>
 <dependency org="objectweb" name="asm-tree"rev="2.2.1" conf="master"/>
</dependencies>
...
```

简单说明一下，`cobertura` 模块定义了三个依赖模块：`asm`、`jakarta-oro` 和 `log4j`，如 清单 5所示。而 `asm` 模块又有一个依赖模块，名为 `asm-tree`，如 清单 6所示。

请注意，图 3 中的 `asm-tree` 目录结构与 图 2中的 `asm` 模块结构相似：

图 3. `asm-tree` 模块的目录结构



当然，差别在于 JAR 文件包含不同的类，图 2 所示的 ivy.xml 文件的定义描述了 `asm-tree` 模块。（碰巧，`asm-tree` 模块未在其 ivy.xml 文件中定义任何依赖项。）

Ivy 进阶

既然您已经掌握了使用 Ivy 的基本知识，下面我将介绍其他一些有用的 Ant 任务。

呈现报告

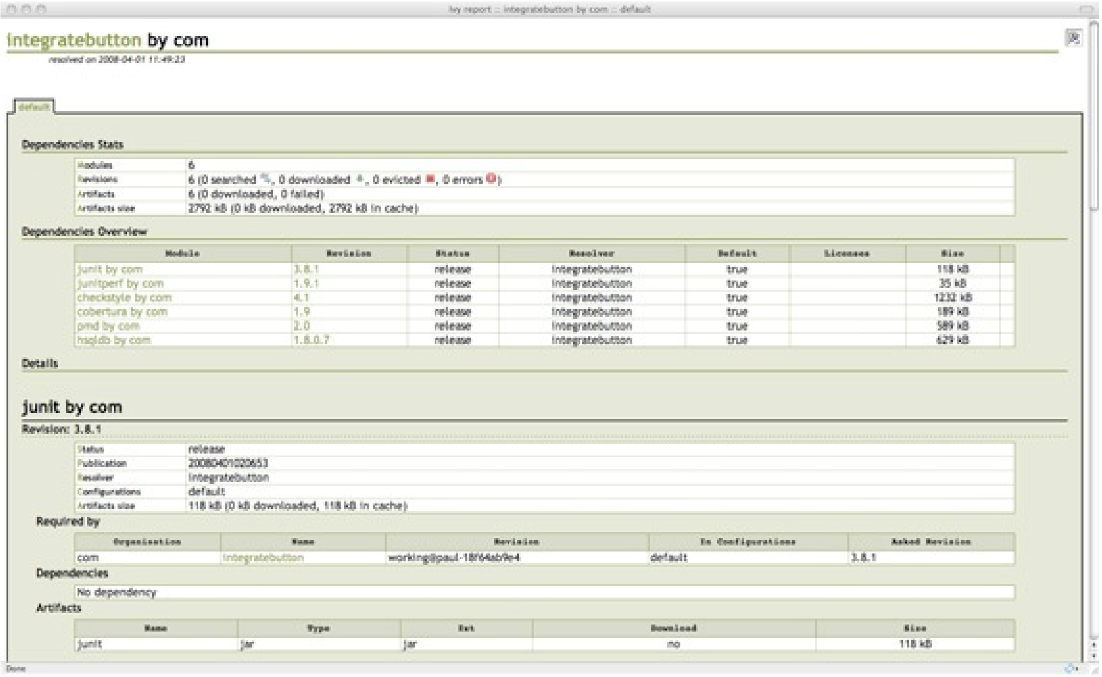
Ivy 提供了一个任务，用于报告一个项目中的依赖文件。清单 7 展示了如何调用 Ivy 的 `report` Ant 任务来创建依赖项列表：

清单 7. 通过 Ant 生成 Ivy 依赖项报告

```
<target name="ivy-report" depends="init-ivy">
 <ivy:report todir="${target.dir}/reports/ivy"/>
</target>
```

清单 7中的脚本生成了一份 HTML 报告，显示了某项目的依赖文件列表。图 4 展示了该报告：

图 4. 显示项目依赖项的 HTML 报告



其他任务

还有其他许多针对 Ivy 的 Ant 任务可供您使用 —通过为 Maven 生成一个 POM 文件来清理本地文件系统缓存。表 1 显示了部分 Ivy 的 Ant 任务及其用途：

表 **1.** 其他 **Ivy Ant** 任务

Task	Purpose
`settings`	对于验证包含存储库的主机最有用
`cachepath`	覆盖本地文件系统上的默认缓存路径，所下载的文件将存放在此路径中
`repreport`	为存储库中的几个模块生成报告
`install`	安装一个模块及其所有依赖项
`makepom`	通过 ivy.xml 文件创建一个 pom.xml file，供 Maven 使用
`cleancache`	清理本地文件系统缓存，强制在下一次构建时从存储库重新检索 JAR 文件

参见 参考资料，了解 Ivy 中可用的其他 Ant 任务。

一切视情况而定

版本控制二进制库

Ivy 并未消除对 JAR 文件进行版本控制的需要。我常常看到有些团队由于得到了可通过 HTTP 访问的存储库，就彻底忘记了将文件置于版本控制系统之中。如果一年之后您需要重新创建软件，而 HTTP 存储库未得到集中管理，重新创建的过程将十分艰难。使用可通过 HTTP 访问的版本控制存储库（如 Subversion）将避免这样的 窘境，因为您可以集中管理 并提供 HTTP 访问

Ivy 集中管理依赖文件，消除了开发团队将 JAR 文件从一个版本控制存储库复制到另一个存储库中时可能出现的膨胀现象。如果您正参与一个简单的项目，将 JAR 文件签入版本控制系统或使用本文开头列出的其他某些技术可能不会显著降低您的速度。但若您的项目规模越来越大，或者您在使用公共文件的企业环境中工作，一种公共方法就变得十分必要。无论是哪种情况，Ivy 都能使定义项目依赖项更为一致、更为可行。因此值得您付出时间研究 Ivy 在您的项目中的应用。

让开发自动化: 自动负载测试

使用 *Apache Ant* 和 *Apache JMeter* 频繁进行负载测试

负载测试通常在开发周期的后期执行，但是并不一定要这样。在 让开发自动化的这一期，自动化专家 Paul Duvall 将向您描述如何创建一个运行 JMeter 测试的预订集成构建，发现和修复开发周期中出现的问题。

您的软件系统可供多少用户同时访问？在不引起性能下降的前提下可以加载多少数据？您的系统有多大的吞吐量需求？间隔多久测试一次这些需求？如果您每天至少可以指定并确认一次这些负载和性能需求得到了满足，又会怎样？通过将负载测试作为预定的自动构建的一部分来运行，您可以更快地确定您的系统在某些负载条件下的执行情况，并快速适应变化。

关于本系列

作为开发人员，我们的工作就是为终端用户实现过程自动化；然而，很多开发人员却忽略了将自己的开发过程自动化的机会。为此，我编写了 让开发自动化这个系列的文章，专门探讨软件开发过程自动化的实际应用，并教您 何时以及 如何成功地应用自动化。

我曾经参与过的一个项目建立了一组很好的自动化测试，可以对应用程序进行负载测试，同时它还可以运行多个事务。问题是，这些测试需要进行一些手动调节，所以开发团队无法在没有人工干预的情况下运行这些测试。这限制了测试器可用时（通常仅工作几个小时）进行测试的次数。在实践中，测试要隔好几天才进行一次 —间隔时间太长，无法及时检测问题。

在本文中，我将探讨如何使用 JMeter 创建自动化测试、将测试作为自动构建的一部分运行，以及将测试设置为每天自动运行（通常当机器的使用率低时）。将测试作为预定构建的一部分运行可以让您：

- 在任何时候执行负载测试
- 在开发过程的初期检测并解决负载和性能问题
- 监视构建服务器的最新的负载测试和性能测试报告
- 减少依靠单个人配置和运行测试时可能出现的瓶颈和错误

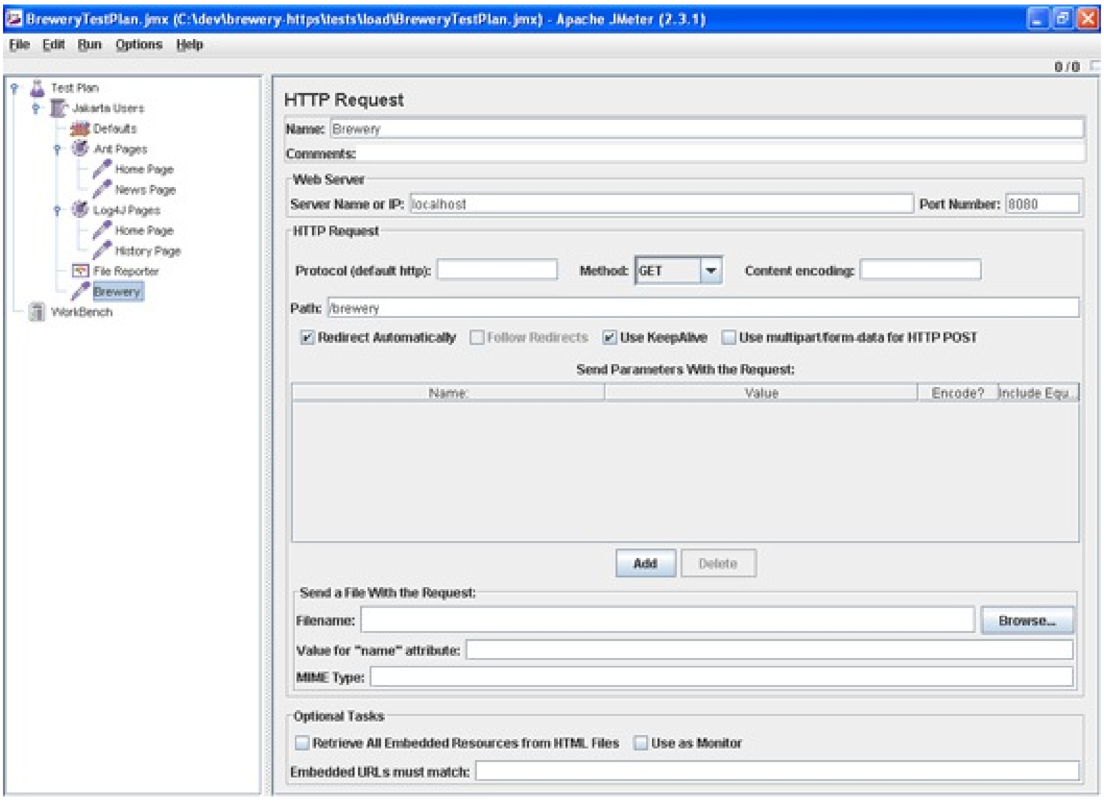
使用 **JMeter** 提升性能

Apache JMeter 是一个开放源码项目，您可以用来在服务器上模拟重负载（有关 JMeter 的更多信息，请参阅 参考资料）。JMeter 的文档集描述了如何使用它的很多功能，并提供了大量例子。

运行 **JMeter**

下载并解压缩 JMeter ZIP 文件（请参阅 参考资料获得下载 JMeter 的链接）之后，使用命令提示符进入您解压缩 JMeter 的位置，并键入 `cd bin` 更改 bin 目录。从 bin 目录键入 `jmeter` 打开 JMeter Swing 应用程序，如图 1 所示：

图 **1. JMeter GUI**



创建测试计划

通过示例编写测试

JMeter 附带了很多示例测试计划和脚本。不必从头创建测试计划，您可以使用 docs 目录中的例子，并随着项目的发展逐步配置测试计划。复杂之处主要在于学习编写可以有效模拟负载和性能需求的负载测试。

您可以使用 JMeter GUI 创建 测试计划。JMeter 中的不同测试计划类型包括：

- Web 测试计划
- 数据库测试计划
- FTP 测试计划
- LDAP 测试计划
- 扩展 LDAP 测试计划
- Web 服务测试计划
- JMS 点对点测试计划
- JMS 主题测试计划

- 监视器测试计划
- 侦听器

每个测试计划都以 XML 格式存储在一个后缀名为 .jmx 的文件中。这种非二进制的格式使以后编辑计划更容易。尽管您可以通过以下 JMeter XML 模式来创建测试计划，但是使用 GUI 要容易得多。稍后您将看到一个例子，该例子用参数表示 JMeter 的配置值，以自定义测试的运行方式。

节省劳力的负载测试

使用 GUI 运行测试，需要一个 人来亲自执行它们。这会增加过程瓶颈和知识筒仓的可能性。通过自动构建（比如 Ant 构建）运行测试，您可以配置 JMeter 测试来运行，而无需打开 JMeter 应用程序。另外，该测试每次都以同样的方式运行 —不会增加额外的工作。

使用 Ant 操作 JMeter 测试

学习了如何使用 GUI 软件工具后，我想看看它是否可以从命令行运行某些实用工具，这样我就不需要反复执行同样的操作。例如，每次打开 JMeter 应用程序时，我喜欢选择 File > Open 来打开文件，然后运行一次或多次测试。我可以为这些组操作编写一个脚本，每次以同样的方式运行它们。幸运的是，已经有人编写了一个 Ant 任务来为 JMeter 做这件事：它执行负载测试，同时提供了一种传入可选参数和属性的方式。

示例构建脚本

JMeter 分区的 extras 目录包含一个示例 build.xml 脚本，说明了 JMeter Ant 任务的用法。

在清单 1 中，我使用 Ant 的 `taskdef` 任务定义 JMeter 任务，我将其命名为 `jmeter`，这样我可以在该 Ant 脚本的其他地方使用它。要使用该脚本，您的 Ant 类路径中必须存在 ant-jmeter.jar 文件（请参阅 参考资料获得下载链接）。

清单 1. 在 Ant 中定义 JMeter 任务

```
<taskdef name="jmeter" classname="org.programmerplanet.ant.taskdefs.jmeter.JMeterTask"/>
```

清单 2中的示例代码运行一个 JMeter 负载测试 BreweryTestPlan.jmx。要运行某个目录中的所有测试，只需输入 `*.jmx`，而不是特定的文件名。`jmeter` 任务所需的属性为 `jmeterhome`、`testplan`（`s`）以及 `resultlog` 或 `resultlogdir`。（清单 2 未显示 `resultlogdir`，因为它使用 `resultlog`。）

清单 **2.** 从 **Ant** 运行 **JMeter**

```
<jmeter
 jmeterhome="${jmeter.home}"
 resultlog="${basedir}/target/JMeterResults.xml">
 <testplans dir="${basedir}/tests/load" includes="BreweryTestPlan.jmx"/>
</jmeter>
```

清单 2 中的 Ant 代码创建了一个名为 JMeterResults.xml 的输出文件，用于创建 HTML 报告。

使用 XSLT 呈现报告

将 JMeterResults.xml 文件作为清单 3 中的 `xslt` Ant 任务的输入，可以为清单 2 中运行的所有 JMeter 测试生成一个 HTML 报告。在 JMeter extras 目录中提供的 XSL 样式表（jmeter-results-detail-report_21.xsl）用于将 JMeterResults 文件转换为 HTML。

清单 **3.** 使用 **XSLT** 创建 **JMeter HTML** 报告

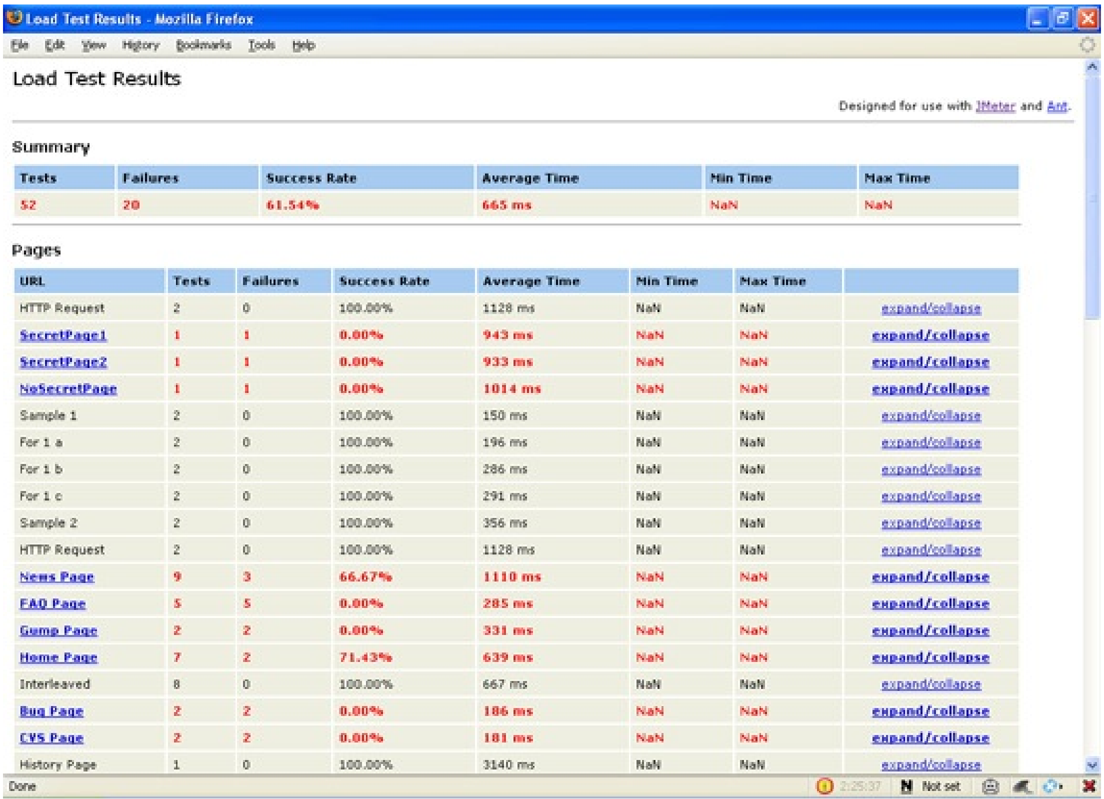
```
<xslt in="${basedir}/target/JMeterResults.xml"
 out="${basedir}/target/JMeterResults.html"
 style="${jmeter.home}/extras/jmeter-results-detail-report_21.xsl"/>
```

JMeter 也提供了一个不太详细的 XSL 样式表文件，用于总结负载测试的结果。

在 HTML 中显示报告

图 2 是一个使用清单 3 中的 `xslt` 任务生成的 HTML 报告的例子。它显示了每个运行的负载测试，以及测试状态、时间和所有测试的聚合状态和时间。

图 **2.** 生成 **JMeter HTML** 报告



Tests	Failures	Success Rate	Average Time	Min Time	Max Time
52	20	61.54%	665 ms	NaN	NaN

Pages

URL	Tests	Failures	Success Rate	Average Time	Min Time	Max Time	
HTTP Request	2	0	100.00%	1128 ms	NaN	NaN	expand/collapse
SecretPage1	1	1	0.00%	943 ms	NaN	NaN	expand/collapse
SecretPage2	1	1	0.00%	933 ms	NaN	NaN	expand/collapse
NoSecretPage	1	1	0.00%	1014 ms	NaN	NaN	expand/collapse
Sample 1	2	0	100.00%	150 ms	NaN	NaN	expand/collapse
For 1 a	2	0	100.00%	196 ms	NaN	NaN	expand/collapse
For 1 b	2	0	100.00%	286 ms	NaN	NaN	expand/collapse
For 1 c	2	0	100.00%	291 ms	NaN	NaN	expand/collapse
Sample 2	2	0	100.00%	356 ms	NaN	NaN	expand/collapse
HTTP Request	2	0	100.00%	1128 ms	NaN	NaN	expand/collapse
News Page	9	3	66.67%	1110 ms	NaN	NaN	expand/collapse
FAQ Page	5	5	0.00%	285 ms	NaN	NaN	expand/collapse
Gump Page	2	2	0.00%	331 ms	NaN	NaN	expand/collapse
Home Page	7	2	71.43%	639 ms	NaN	NaN	expand/collapse
Interleaved	8	0	100.00%	667 ms	NaN	NaN	expand/collapse
Bug Page	2	2	0.00%	186 ms	NaN	NaN	expand/collapse
CVS Page	2	2	0.00%	181 ms	NaN	NaN	expand/collapse
History Page	1	0	100.00%	3140 ms	NaN	NaN	expand/collapse

稍后我将在本文中向您展示如何从 CruiseControl Continuous Integration (CI) 服务器（请参阅参考资料）中显示这些报告。

向 JMeter 传递参数

根据您运行的测试类型，您可能想要传递参数和属性，以改变单个测试或一组测试执行的方式。例如，清单 4 展示了如何增加 JVM 内存并指定线程和循环的数量：

清单 **4.** 向 **JMeter** 传递可选参数和属性

```
<jmeter
 jmeterhome="${jmeter.home}"
 resultlog="${basedir}/target/JMeterResults.xml">
 <jvmarg value="-Xincgc"/>
 <jvmarg value="-Xmx128m"/>
 <jvmarg value="-Dproperty=value"/>
 <property name="request.threads" value="5"/>
 <property name="request.loop" value="50"/>
 <property name="jmeter.save.saveservice.assertion_results" value="all"/>
 <property name="jmeter.save.saveservice.output_format" value="xml"/>
 <testplans dir="${basedir}/tests/load" includes="BreweryTestPlan.jmx"/>
</jmeter>
```

可以使用很多内置的其他参数和属性来修改 JMeter 测试运行的方式（有关详细信息，请参阅参考资料）。

在执行负载测试的方式上，使用参数和属性提供了一定的灵活性，但是它不能解决如何在不同的目标环境中运行负载测试的问题，比如测试和验证环境。要向测试计划添加特定于环境的信息，您需要在 .jmx 文件中放入一些记号，以便当负载测试在自动构建脚本中运行时可以

对 .jmx 文件进行过滤和修改。

及时负载测试

使用自动构建运行负载测试时，将其安排为按某个周期运行，比如每晚运行一次。您可以使用 CI 或构建管理服务器来实现。

安排 CruiseControl 每天运行负载测试

使用 CI 服务器的目的在于，只要向项目的版本控制存储库应用了更改，就运行一个自动构建。您也可以将其配置为按特定次数运行构建。由于负载测试通常需要较多的计算资源，在这些资源未被占用时运行测试（例如深夜或清早）会比较好。

在 清单 5中，一个自动构建被安排在晚上 11:00 点 (2300) 使用 CruiseControl（请参阅 参考资料）运行。您可以修改 CruiseControl 配置文件，以使用一个特定的 Ant 目标运行一个委托构建，比如一个给定的 `run-load-tests` 构建。

清单 **5.** 使用 **CruiseControl** 运行预定的负载测试

```
...
<modificationset>
  <svn RepositoryLocation="${svnrepo.location}"/>
  <timebuild username="admin" time="2300"/>
</modificationset>
...
```

通过将负载测试安排在晚上运行（如清单 5 中一样），您将不会听到有关加班、休假或忘记运行测试等借口 —它们会自动运行。

在 CruiseControl 中显示报告

您已经看到了如何使用 Ant 显示 JMeter 测试报告。但是，JMeter 报告只能与单个机器上的一个开发人员通信。负载测试会影响整个应用程序，所以整个团队都会希望看到结果。好处在于，您可以轻松配置您的 CI 服务器，以显示这些报告。因为已经使用 Ant 生成了这些报告，所以只需要使 JMeter HTML 报告可以从 CruiseControl 项目仪表板访问。您可以向 CruiseControl 的 config.xml 文件添加几行代码来实现这个目的，如清单 6 所示：

清单 **6.** 配置 **CruiseControl** 来显示 **JMeter** 报告

```
<project name="brewery">
...
<log>
 <merge dir="merge dir="projects/${project.name}/reports/jmeter" />
</log>
...
</project>
```

现在，团队中的每个人都可以（真正地）共享这些信息了。很多其他 CI 和构建管理服务器也提供类似的报告集成功能。

结束语

在本文中，我展示了如何向您的开发工具箱添加自动化负载测试。通过使用自动构建运行负载测试，然后将测试安排为定期运行，您可以在系统容量问题出现之前及时发现它们。这种方法使得评估架构和数据更改的影响变得更加容易。当与本文章 系列 中描述的其他技术结合使用时，开发团队常常能够交付更高质量的软件。

下载

描述	名字	大小
本文的示例 Ant 脚本	j-ap04088-jmeter-example.zip	6KB

让开发自动化: 使用自动化加速部署

利用自动化加速软件在不同环境间的迁移

自动化构建不仅仅适用于开发团队 —— 在将软件从开发迁移到生产这一过程中也大有作为。在这一期 让开发自动化中，自动化专家 Paul Duvall 将介绍如何结合使用 Ant 和 Java™Secure Channel 将软件远程部署到多个目标环境中。

关于本系列

作为开发人员，我们的工作就是为终端用户实现过程自动化；然而，很多开发人员却忽略了将自己的开发过程自动化的机会。为此，我编写了 让开发自动化这个系列的文章，专门探讨软件开发过程自动化的实际应用，并教您 何时以及 如何成功地应用自动化。

您是否曾注意到，很多团队总是在将软件从开发环境迁移到生产环境之后才想到改善？我曾经遇到过一些团队，他们的开发周期长达几个星期甚至几个月不等 —我认为这是浪费时间。为什么不像自动化构建一样，通过自动化大幅度减少花费在部署配置问题上的时间，并因此提升基础架构的效率？

仔细想一想：软件部署流程中的低效率意味着将推迟应用程序到用户的交付。更糟糕的是，一些人认为，对于大多数项目，部署类似于撕掉创可贴（仅会产生暂时的疼痛），然而，部署问题将一再滞留，并且在每次交付时频繁出现。

除了延迟交付外，低效的部署基础架构降低了团队对软件修改的适应性，这使得他们经常将过多的功能全部填充到一个版本中（因为版本不会经常发布）。这导致了恶性循环：企业希望尽快将软件交付给用户，但是这个流程太长，因此所有人都力求设计一个全面的版本（big bang）来最大化业务机遇。

实现无忧部署

基本部署流程包括编译、数据修改集成（例如数据库表）、向其他计算机远程部署发行包（如 JAR 和 WAR），以及管理远程计算机中的资源。尽管如此，仍然有很多工作可以通过自动化完成，比如创建安装媒体、测试、生成用户文档等等。在本文中，我将介绍这些基本内容并演示如何将这些流程纳入到自动化构建流程中。具体来说，您将了解以下流程：

- 向远程机器部署二进制文件
- 外部化（externalize）配置属性
- 远程更新 MySQL RDBMS
- 远程配置 Jakarta 的 Tomcat Servlet 容器

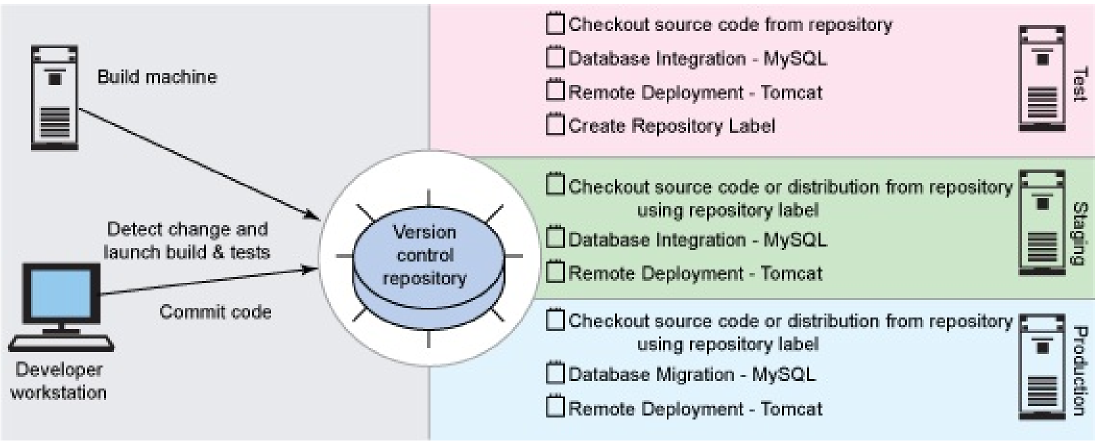
通过自动化实现这些流程，您能够更迅速更顺利地向用户交付软件。

必要工具

实现自动化部署的核心工具是一个构建脚本；在本文中，我使用的是 Ant。我的 Ant 脚本将使用属性文件（特定于目标环境，如演示和生产环境），通过 Ant 的 `sql` 任务与 MySQL 数据库交互，使用 Java Secure Channel (JSch) 将文件复制到远程机器上（通过 Secure-Copy 协议（SCP））并停止和启动 Tomcat 服务（通过 SSH）。

图 1 演示了这一流程的高级架构视图。关键的一点是，所有软件资源均存储在一个版本控制库中，因为理想的构建流程首先将对源代码（以及配置文件）进行签出，并在本地构建和打包组件，随后远程执行 SQL 语句，最后部署发行包并重新启动 Tomcat。

图 **1.** 实现远程部署的高级构建架构



您可以自动执行所有这些流程，这样，通过执行一条命令或单击一下鼠标就可触发部署，甚至不需要人为干涉就可对流程进行调度。是不是很令人兴奋？

外部化属性

对于不同的目标环境，配置值（例如文件位置、主机名、数据库名和端口号）可能各不相同，因此不能进行硬编码（例如在源代码中）。这些属性在 *.properties*文件中得到了完善的管理。通过外部化属性，可以使用同一个构建脚本在一个环境中编译，然后在另一个环境中部署，而不需要修改或重新编译源代码。

属性规则

在转换到不同环境时，如果需要修改某个值，那么将这个值放到外部的 .properties 文件。如果构建脚本中有多处引用该值，那么应将其转化为构建脚本（即 build.xml）的属性。如果 始终只存在一个引用，就没有必要将其转化为属性（但这仅仅是一个假定条件）。

清单 1 演示了在 Ant 构建脚本中定义属性的简单示例，它允许您将一个 .properties 文件作为系统参数传递（例如 `test.properties` ），其中包含针对特定目标环境的所有值。`property.file.location` 可解析为诸如 C:\Documents 和 Settings\patrick.henry\test.properties 这样的路径。例如，您可以在命令行输入：`ant -Dproperty.file.location=C:\Documents and Settings\patrick.henry\test.properties` 。

清单 **1.** 外部化 **property** 属性

```
<property file="${property.file.location}" />
```

清单 2 显示了一个示例目标环境 .properties 文件。文件中的 *values*属性在不同目标环境中应该（或者可以）是不同的，但是 *names*属性则保持不变。

清单 **2. property** 文件中的示例属性和对应的值

```
db.database=brewery
db.username.system=root
db.password.system=sa
db.username=root
db.password=sa
db.hostname=my-hostname.domain.com
db.driver=com.mysql.jdbc.Driver
db.port=3306
db.url.system=jdbc:mysql://${db.hostname}:${db.port}/
db.url=jdbc:mysql://${db.hostname}:${db.port}/${db.database}
```

通过外部化 property 属性和值，我们可以创建一个更加灵活的构建和部署架构，从而支持多个目标环境。

以简单性为核心

向另一个环境部署软件的流程不应该过于繁杂；应尽量保持简单，就好象输入 “deploy” 一样。幸运的是，诸如 Ant 这样的构建系统使这成为了现实。通过在逻辑上定义一个工作流，按顺序执行一系列步骤，您可以创建一个简单的调用命令。

清单 3 的 `depends` 属性中枚举的 Ant 目标从较高层次定义了最佳自动部署流程。首先，脚本从本地环境中删除以前生成的工件（使用 `clean` 目标）、编译源代码、远程创建数据库、应用测试数据、启动数据库，最后将 WAR 文件远程部署到位于目标环境中的 Tomcat 容器中。

清单 **3.** 远程部署中执行的关键目标

```
<target name="build"
 depends="clean, compile, refresh-database, remote-tomcat-deploy" />
```

刷新数据库并进行远程部署并非易事；然而，通过使用一些聪明的脚本，一切都将变得简单！

自动化 DBA

在设置目标测试环境时，经常需要执行一些手动操作，例如配置数据库、插入测试数据、删除旧条目以及其他重复性（并且容易出错）流程。幸运的是，在部署期间处理数据库可以变得更加简单。

Data Definition Language（DDL）语句（如删除现有数据库、创建数据库和创建数据库用户）以及 Data Manipulation（DML）语句（如 `insert` 语句）可以轻松地脚本化并作为 Ant 构建脚本的一部分运行。而且，还可以远程执行这些语句。

例如，通过从目标环境 .properties 文件传递一个 `db.url.system` 属性（如清单 4 所示），构建环境可以针对一个远程数据库执行 SQL 语句：

清单 **4.** 创建数据库和插入数据的脚本

```
<target name="refresh-database" depends="create-database,insert-data" />
<target name="create-database">
 <sql driver="${db.driver}"
   url="${db.url.system}"
   userid="${db.username.system}"
   password="${db.password.system}"
   src="${database.dir}/create-database.sql">
   <classpath>
     <pathelement location="${mysql-connector.jar}"/>
   </classpath>
 </sql>
</target>
...
<target name="insert-data">
 <sql driver="${db.driver}"
   url="${db.url}"
   userid="${db.username}"
   password="${db.password}"
   src="${database.dir}/insert-data.sql">
   <classpath>
     <pathelement location="${mysql-connector.jar}"/>
   </classpath>
 </sql>
</target>
```

清单 5 中 insert-data.sql 文件的内容是从清单 4 的 `insert-data` 目标调用的。任何 SQL 语句，DDL 或 DML，都可通过 Ant 的 `sql` 任务以类似方式执行。

清单 **5.** 执行数据插入的 **SQL** 语句

```
insert into beer(id, beer_name, date_received) values
 (1, 'Samuel Adams Lager','2006-12-09');
insert into beer(id, beer_name, date_received) values
 (2, 'Guinness Stout','2006-12-29');
insert into beer(id, beer_name, date_received) values
 (3, 'Olde Saratoga Lager','2007-02-14');
insert into beer(id, beer_name, date_received) values
 (4, 'Sierra Nevada Pale Ale','2007-05-14');
```

现在我已经更新了远程数据库，下一个逻辑步骤是向运行 Tomcat 的远程环境部署一些资源。

发行和部署

远程部署和本地部署在实现方面并非完全不同，它仅需要一个不同的通道，从而将资源安全地从一个位置复制到另一个位置（从构建机器复制到目标环境）。在大多数企业中，安全性至关重要，因此仅仅使用 FTP 和 telnet 并不能满足需求。在这种情况下，SCP 和 SSH 可以轻松完成任务。通过 Ant 可以很方便地使用这些通道；事实上，我经常使用 JSch 中的 `sshexec` 和 `scp` 任务远程复制文件并在远程机器上运行命令。

迁移到生产环境

尽管某些软件应用程序（例如，Software as a service，即 *SaaS*）可以改变部署频率，但是仍然难于迁移到生产环境中。您需要添加应用程序和数据库回滚，确保软件系统能够回滚到以前的状态。这一点至关重要，因为有可能带来数百万美元的损失或盈利。和测试软件系统本身一样，必须对自动化部署流程进行严格测试。

使用 **SCP** 安全复制文件

SCP 能够在两台机器之间安全复制资源。很多工具都支持 SCP。在 Ant 中，理论上讲，在 Ant 中，JSch 将使用 SCP 复制资源（如 JAR 文件），而 不需要人为干预，我喜欢将其称之为 自动化。

在清单 6 中，`scp` 任务（JSch 提供）将构建机器中的 WAR 文件复制到远程机器上。JSch 库（jsch-0.1.36.jar）必须位于 Ant 的类路径中，以利用 `scp` 任务。

清单 **6.** 将一个 **WAR** 文件从一个机器中安全复制到另一个机器

```
<target name="copy-tomcat-dist">
 <scp file="${basedir}/target/brewery.war"
 trust="true"
 keyfile="${ssh.key.file}"
 username="${ssh.username}"
 passphrase=""
 todir="${ssh.server.username}:${ssh.server.password}@${ssh.server.hostname}
 :${tomcat.home}/webapps" />
</target>
```

当调用 `scp` 任务时，需要提供进行复制的本地文件的位置，以及本地 SSH 私钥文件的位置（清单 6 中的 `ssh.key.file`，用于安全身份验证）。最后，需要在远程机器上提供一个位置（清单 6 中的 `ssh.server.hostname`），`scp` 将把（一个或多个）本地文件放在这个位置。

使用 SSH 远程调用流程

与使用 SCP 一样，在远程机器上运行命令通常需要某种安全机制，例如 SSH。在清单 7 中，我使用 JSch `sshexec` Ant 任务停止和重新启动位于某个远程机器上的 Tomcat 容器。构建流程刚刚将一系列资源（如 WAR 文件）复制到这台机器上。

清单 **7.** 停止和重新启动远程 **Tomcat** 实例

```
<target name="remote-tomcat-stop>
 <sshexec host="${ssh.hostname}"
 port="${ssh.port}"
 keyfile="${ssh.key.file}"
 username="${ssh.username}"
 passphrase=""
 trust="true"
 command="${tomcat.home}/bin/shutdown" />
 <sleep seconds="${sleep.time}" />
</target>
...
<target name="remote-tomcat-start">
 <sshexec host="${ssh.hostname}"
 port="${ssh.port}"
 username="${ssh.username}"
 passphrase=""
 trust="true"
 keyfile="${ssh.key.file}"
 command="${tomcat.home}/bin/startup" />
 <sleep seconds="${sleep.time}" />
</target>
```

在清单 7 中，我提供了托管 Tomcat 的机器的名称，Tomcat 的端口号（通常为 8080）、私钥文件（`ssh.key.file`），这样，构建脚本可以安全地访问这个机器并执行特定命令。在本例中，可以看到，我依次调用了 `shutdown` 和 `startup` 命令。

理论上讲，完成这一步骤后，我已经完成了下面这些任务：配置远程数据库、将一个 Web 应用程序移至远程计算机、运行一个 Tomcat 实例。至此，人们可以正常测试甚至使用新版本的应用程序。

结束语

希望本文已经向您展示了如何轻松实现自动化部署流程。将软件从开发环境中交付到用户手中不能（或不应该）是一个手动流程，也不需要将它从开发团队的构建流程明确分离出来。实际上，通过本文介绍的方法，软件发行可以像按下某个按钮一样简单，当然，这必然会显著提高开发团队 频繁交付特性的能力。

让开发自动化: 持续集成反模式

通过避免反模式轻松实现持续集成

尽管持续集成（Continuous Integration，CI）可以非常有效地减少项目的风险，但是它对与编程相关的日常活动提出了很高的要求。在这一期 让开发自动化中，自动化专家和 *Continuous Integration: Improving Software Quality and Reducing Risk*的作者之一 Paul Duvall 列举了一系列 CI 反模式并解释了如何避免它们。

在我的职业生涯中经常发现，通过了解在特定情况下 不应该做什么，可以学到更多知识。例如，在我职业生涯的早期，由于需要快速发布软件，我省略了单元测试，因为我认为不值得做这些工作。幸运的是，我已经学到 绝不应该将未经测试的代码投入生产；因此开始坚持编写单元测试。

整个 IT 行业似乎都主要采用这种学习方式；实际上，我们甚至专门创建了 反模式（*anti-pattern*）这个词，表示在特定环境中不应该采用的做法。反模式是看起来似乎有好处，但是最终可能产生严重影响的解决方案。

看似真实的假象

遗憾的是，我发现当缺少经验的团队试图采用 CI 时，他们很可能错误地采用许多反模式，这最终导致他们不但没有获得预期的好处，反而遇到一大堆麻烦。不幸的是，在这种情况下，团队常常将麻烦归罪于 CI 本身。因此，我常常听到 “CI 不适合大项目” 或 “我们的项目 太特殊，不适合采用 CI” 这样的说法，实际上 CI 根本不是问题的原因 —是某些做法的不恰当应用或者缺少某些方法导致了这些麻烦。

关于本系列

作为开发人员，我们的工作就是为终端用户实现过程自动化；然而，很多开发人员却忽略了将自己的开发过程自动化的机会。为此，我编写了 让开发自动化这个系列的文章，专门探讨软件开发过程自动化的实际应用，并教您 何时以及 如何成功地应用自动化。

在本文中，我要描述与 CI 相关的六个反模式：

- 签入不够频繁，这会导致集成被延迟
- 破碎的构建，这使团队无法转而执行其他任务
- 反馈太少，这使开发人员无法采取纠正措施
- 接收 垃圾反馈，这使开发人员忽视反馈消息
- 所拥有的 机器缓慢，这导致延迟反馈
- 依赖于 膨胀的构建，这会降低反馈速度

如果您采用 CI 的时间足够长，那么几乎肯定体验过这些反模式的效果。这没关系，但是如果它们发生得太频繁，就会大大限制 CI 的好处。因此，如果您希望避免这些反模式并控制它们的负面影响，那么本文正适合您。

由于签入不够频繁导致的延迟集成

名称：签入不够频繁

反模式：由于所需的修改太多，源代码长时间签出存储库。

解决方案：频繁地提交比较小的代码块。

实施 CI 的前提是团队可以快速获得关于当前开发的代码的反馈；而且，与传统的集成相比，这种频繁的软件集成风格会减少集成花费的时间（和麻烦）。但是，有效的 CI 假设修改会频繁地发生（所以可以频繁地执行构建！）。如果代码长期留在开发人员的桌面（而不是存储库）中，那么就会出现糟糕的情况，因为在系统的不同部分中会出现其他修改。

每天提交一次，成功实现集成

一条常用的经验规则是 至少每天签入一次代码。我使用一种有效的技术：如果我觉得需要把工作停一下，就会先看看目前是否可以运行本地构建，然后提交代码。然后我才会暂停工作。

从本质上说，如果不频繁地提交修改，集成就会延迟；延迟越长，消除其严重影响就越困难（比如其他人的修改可能会影响您的代码）。对于使用 CI 的项目，我建议开发人员 至少每天签入一次代码，但是我相信最好是每天签入多次。

任务越小，工作越轻松

我常常听到一些开发人员抱怨说，他们要忙于修改那么多文件，哪有精力每天签入代码。实际上，这正是我要说的要点 —为了每天提交源代码修改，需要将任务划分得更小。实际上，需要将编程任务划分成小块，这样修改也会更小。

不要在一个大任务中实现一个业务对象上的所有特性，例如编写 `read()` 、`write()` 、`update()` 和 `delete()` 方法的原型；而是应该首先编写 `read()` 方法（以及对应的测试），然后签入这个类，从而与整个代码基集成。接下来，可以实现另一个方法，再次执行签入，直到完成整个任务。这样的话，就可以让 CI 的好处最大化，而且会让您确信自己的代码可以与 别人的代码相互配合。

请记住，即使您和您的团队正确地执行许多 CI 实践，如果团队成员不坚持至少每天签入一次源代码修改，那么 CI 的好处会大打折扣。这常常会让人误以为 CI 是无效的，这种想法实在大错特错。

破碎的构建减慢了开发的节奏

名称：破碎的构建

反模式：构建长时间破碎，导致开发人员无法签出可运行的代码。

解决方案：在构建破碎时立即通知开发人员，并以最高优先级尽快修复破碎的构建。

无论您信不信，构建破碎的时间越长，就越麻烦。这是因为文件、修改和依赖项越多，隔离缺陷就越困难。因此，当通知某人构建破碎时（通过电子邮件、RSS 或其他机制），他应该优先解决这个问题；否则，构建破碎的时间越长（尤其是对于频繁修改代码的团队），就越难纠正。

破碎的构建不总是坏事。实际上，破碎的构建会让您迅速地意识到软件出了问题。当构建 频繁地破碎或长时间破碎时，破碎的构建就会成为问题。在构建破碎的情况下，绝不应该置之不理。

不存在永不破碎的构建

我曾经听到一些开发人员说，“永远不要出现破碎的构建！” 这是个糟糕的建议。我们希望避免许多常见的构建错误，比如缺少文件或破碎的测试；但是，不破碎的构建也可能反映一些问题。构建可能只做少量工作（可能只是执行一个编译和几个单元测试）。我称之为 “持续忽视（Continuous Ignorance）”，这种情况有时候比频繁的破碎构建更糟糕。

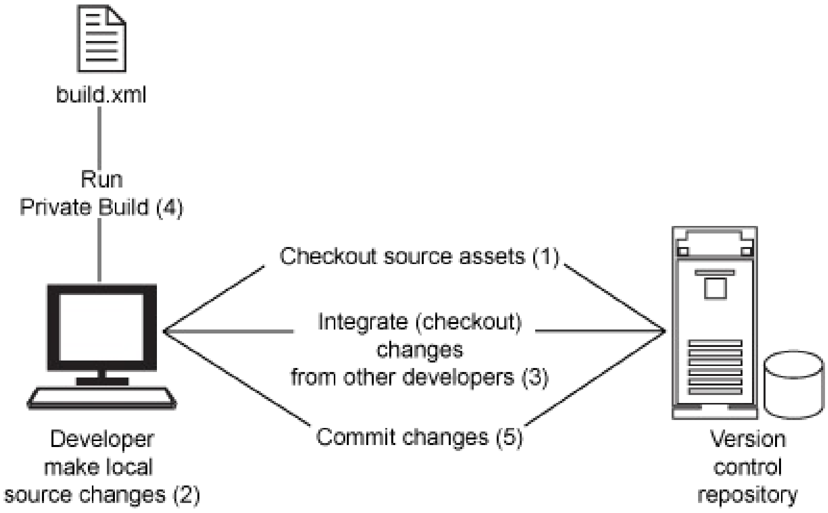
用私有构建减少破碎的构建

防止破碎构建的有效技术之一是，在将代码提交到存储库之前，运行 私有构建（*private build*）。执行私有构建的步骤如下：

1. 从存储库签出代码。
2. 在本地修改代码。
3. 用存储库执行更新，从而集成其他开发人员所做的修改。
4. 运行本地构建。
5. 构建成功之后，将修改提交到存储库。

图 1 说明了这种做法。注意，这个工作流强调频繁地与存储库执行同步，从而保证定期签入并限制破碎的构建 —这真是一举两得！

图 1. 运行私有构建减少破碎的集成构建



通过在签入源代码之前执行私有构建（当然是频繁地执行），可以避免许多导致构建破碎的典型错误；因此，可以节省时间和减少麻烦。

由于反馈太少，无法采取纠正措施

名称：反馈太少

反模式：团队没有把构建状态通知发送给团队成员；因此，开发人员不知道构建已失败。

解决方案：使用各种反馈机制传播构建状态信息。

在设置 CI 系统时，团队常常认为接收电子邮件是浪费时间；因此，他们决定不发布通知。但是，如果没有对构建的反馈，就无法采取纠正措施。实际上，反馈是 CI 最重要的方面之一；因此，反馈是否 有效也非常关键。

如果希望扩展将构建状态信息发布给团队成员的机制，那么使用视觉和声音设备可能很有帮助，对于集中工作的团队尤其如此。Ambient Orb 这样的设备可以几乎实时地反映构建的状态。例如，当构建失败时，orb 可以显示红色；当构建通过时，orb 显示绿色。另外，orb 还可以传播其他信息，例如通过改变颜色表示代码库的复杂性是增长还是下降（比如绿色表示良好，黄色表示糟糕）。

发挥创造力

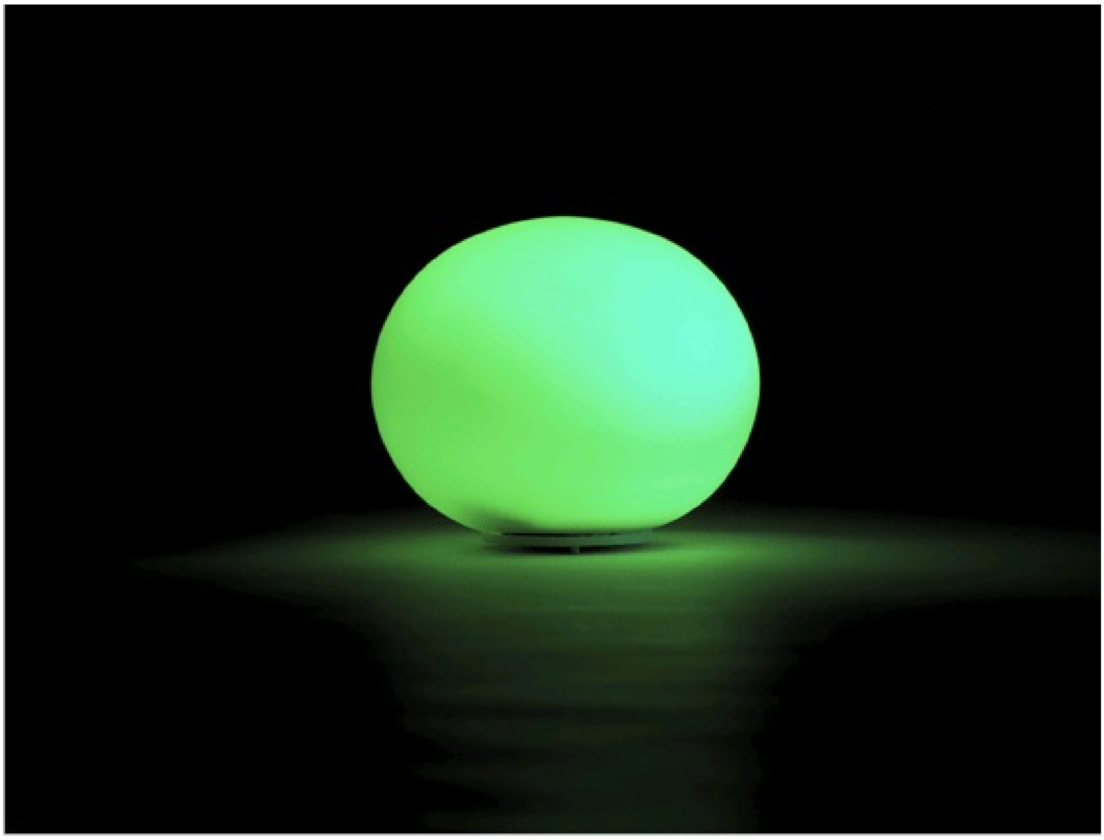
设置 Ambient Orb 非常容易。清单 1 演示如何使用 Quality Lab 的开放源码软件 `OrbTask` 在 Ant 中设置 Ambient Orb：

清单 **1.** 使用 **Ambient Orb Ant** 任务

```
<target name="notifyOrb" >
 <taskdef classname="org.qualitylabs.ambientorb.ant.OrbTask"
   name="orb" classpathref="orb.class.path"/>
     <orb query="http://myambient.com:8080/java/my_devices/submitdata.jsp"
      deviceId="AAA-9A9-AA9"
      colorPass="green"
      colorFail="red"
      commentFail="Code+Duplication+Threshold+Exceeded" />
 </target>
```

清单 1 中的任务对于通过状态将 orb 改为绿色，对于失败状态显示红色。图 2 显示绿色的 orb，这表示最近的构建状态是成功：

图 **2.** 成功的构建！



团队可以创造性地使用各种反馈机制，让团队成员不会忽视构建状态消息。另外，这些技术也会让 CI 变得生动有趣，让人们更容易注意到需要采取措施的问题。

其他通知机制包括：

- RSS feed
- 任务栏监视器，比如 CCTray（用于 CruiseControl）
- X10 设备（比如 LavaLamps）
- 通过 Jabber 等发送的即时消息
- SMS（Text Messages）

警告：需要在信息过多和信息过少之间找到一个平衡点。反馈机制应该随着工作环境定期调整。例如，对于集中工作的团队，声音提示可能是有效的（比如在构建失败时发出火灾警报）；但是，其他团队可能更喜欢 Ambient Orb（它不会在您陷入沉思时吓着您）。

垃圾反馈

名称：垃圾反馈

反模式：团队成员很快被构建状态消息淹没（成功、失败或界于这两者之间的各种消息），所以开始忽视这些消息。

解决方案：反馈要目标明确，使人们不会收到无关的信息。

与 “反馈太少” 反模式相反，我常常发现团队天真地认为，当 CI 服务器做任何事情时，每个人都应该接到反馈（比如电子邮件）。信息一旦泛滥，人们就会忽视它们；如果反馈太多，团队很快就会将 CI 反馈看做垃圾。所以，当发生真正严重的问题（比如构建真的破碎了）时，可能无法引起注意。

通过精确地确定反馈的目标，尽可能减少垃圾反馈

清单 2 给出一个 CruiseControl 配置文件示例，演示如何有效地使用电子邮件通知。在这个示例中，无论构建是成功还是失败，技术主管都会收到电子邮件，项目经理只在构建失败时收到电子邮件，最近向存储库提交源代码修改的开发人员也会收到通知。

清单 **2.** 使用 **CruiseControl** 发送电子邮件通知

```
<project name="brewery">
...
<publishers>
 <htmlemail
   css="./webapps/cruisecontrol/css/cruisecontrol.css"
   mailhost="localhost"
   xsldir="./webapps/cruisecontrol/xsl"
   returnaddress="cruisecontrol@localhost"
   buildresultsurl="http://localhost:8080"
   mailport="25"
   defaultsuffix="@localhost" spamwhilebroken="false">
   <always address="techlead@localhost"/>
   <failure address="pm@localhost" reportWhenFixed="true"/>
 </htmlemail>
</publishers>
...
```

反馈是 CI 系统最重要的方面之一，值得好好讨论一下。反馈太少和 垃圾反馈是两个极端，要在它们之间找到一个适当的平衡点。当构建破碎时，必须及时地将反馈发送给适当的人，而且必须提供采取纠正措施所需的信息。如果构建是成功的，那么应该只向少数人发送反馈，

包括最近提交修改的开发人员以及希望掌握最新情况的技术领导。如果不加区分地把所有状态消息发送给所有人，肯定会大大损害 CI 过程的效果。

不要让缓慢的机器导致反馈延迟

名称：缓慢的机器

反模式：用一台资源有限的工作站执行构建，导致构建时间太长。

解决方案：增加构建机器的磁盘速度、处理器和 RAM 资源，从而提高构建速度。

几年前，我参与了一个相当大的项目，它有超过一百万行代码，编译需要花两个小时以上。当我们试图更频繁地执行集成时，等待持续管理团队执行集成的时间越来越长，让人很不耐烦。当然，两个小时其实是最好的情况，因为构建常常失败，所以这个过程常常要花几 天（真是痛苦啊！）。这种情况持续几周之后，解决方案就非常明确了：必须购买一台更强大的机器，它的磁盘空间必须足以容纳所有要签出的文件和构建生成的文件，它的处理器速度更快，可以处理许多指令，RAM 数量要足以运行测试和其他需要大量内存的进程。

您觉得需要提高速度吗？

有了这台强大的机器，我们将构建时间从两个小时降低到了 30 分钟；所以，通过花费额外资金购买最新的机器，我们节省了大量时间和金钱，而且最终能够更快地集成软件（这意味着更快地发现问题！）。

用更强大的工作站执行集成构建总是没错的；但是，这个故事的要旨在于，如果发现构建机器在速度、内存或硬盘方面无法让人满意，就应该认真考虑进行升级；加快构建可以帮助我们更快地获得反馈，快速纠正问题，更快地转到下一个开发任务。

膨胀的构建导致反馈延迟

名称：膨胀的构建

反模式：把太多的任务添加到提交构建过程中，比如运行各种自动检查工具或运行负载测试，从而导致反馈被延迟。

解决方案：一个构建 管道（*pipeline*）可以运行不同类型的构建。

一些开发团队喜欢把能添加的所有过程都添加到自动构建中，但是他们忘了执行这些操作是要花 时间的。还记得吗？我遇到的那个项目的编译要花两个小时。想像一下，如果把执行测试添加到构建过程中，会怎么样呢？对于一百万行代码，您认为运行静态分析工具要花多长

时间？如果您认为耗时八小时的构建过程是难以令人置信的，那么再想想吧。我经常遇到这种情况。

为了向团队成员提供更多的构建信息，团队往往会逐渐增加构建过程的内容。要向开发团队提供快速反馈，还要从 CI 构建过程提供有用的信息，必须在这两个目标之间取得平衡。

通过构建管道提高效率

如果发现构建过程耗时太长，而且已经实现了其他改进技术（比如改用更快的机器）并优化了测试执行时间，那么就有必要考虑创建所谓的 构建管道（*build pipeline*）。构建管道的用途是异步地执行长时间运行的过程，这样的话，开发人员签入代码之后，不需要长时间等待反馈。

例如，如果执行一个构建过程要花 10 分钟以上，那么可以创建一个构建管道，在某人将代码提交到存储库之后，它会运行一个初步的轻型构建。这个 “提交” 构建由编译和运行快速单元测试等轻型过程组成。如果这个初步构建成功了，就可以运行第二个构建，它执行长时间运行的测试、软件检查，甚至包括部署到应用服务器上。

例如，在清单 3 中，让 CruiseControl 检查存储库中的修改。当发现修改时，CruiseControl 运行一个所谓的 委派（*delegating*）构建，它调用项目的主构建文件（在使用 Ant 时是 build.xml）。但是，特殊之处是 CruiseControl 执行另一个目标，这个目标执行一些轻型过程，比如编译和细粒度的单元测试。

清单 **3.** 检查修改的 **CruiseControl** 配置

```
<project name="brewery-commit">
...
 <modificationset quietperiod="120">
   <svn RepositoryLocation="http://brewery-ci.googlecode.com/svn/trunk"/>
 </modificationset
...
```

在清单 4 中，CruiseControl 检查对 `brewery-commit` 项目的修改（这个项目不在存储库中 — 它实际上查看一个日志文件）。当发现修改时，CruiseControl 运行另一个委派构建。这个构建调用相同的构建文件，但是执行另一个目标。这个目标可能执行长时间运行的过程，比如功能测试、软件检查等等。

清单 **4.** 执行长时间运行的构建的 **CruiseControl** 配置

```
<project name="brewery-secondary">
...
 <modificationset quietperiod="120">
   <buildstatus logdir="logs/brewery-commit"/>
 </modificationset>
...
```

膨胀的构建反模式是导致 CI 无法实施的最常见原因。但是，正如您看到的，使用构建管道可以避免这种情况。有效的构建管道应该充分利用 “80/20” 规则：百分之 20 的构建时间花费在导致百分之 80 的构建错误（比如缺少文件、破碎的编译和测试失败）的部分上。完成这个过程之后，开发人员接到反馈，然后运行第二个构建过程，这个过程的运行时间比较长，但是只产生百分之 20 的构建错误。

反模式是可以纠正的

CI 反模式会妨碍团队从持续集成实践中获得最大的收益；但是，本文描述的技术有助于限制这些反模式发生的频率。这些技术包括：

- 经常提交代码，可以防止集成变得复杂。
- 在提交源代码之前运行私有构建，可以避免许多破碎的构建。
- 使用各种反馈机制避免开发人员忽视构建状态信息。
- 有针对性地向可以采取措施的人发送反馈，这是将构建问题通知团队成员的好方法。
- 花费额外资金购买更强大的构建机器，从而加快向团队成员提供反馈的速度。
- 创建构建管道来缓解构建膨胀。

本文描述了我最常遇到的一些反模式，但是还有其他反模式，包括：

- 持续忽视（**Continuous Ignorance**），也就是构建过程只包含很少的过程，导致构建总是成功。
- 构建 只在您的机器上执行，这会延长引入缺陷和纠正缺陷之间的时间。
- 瓶颈提交（**Bottleneck Commits**），这会导致破碎的构建，让团队成员无法回家。
- 运行 间歇构建（**intermittent build**），这使反馈延迟。

明年我会讨论其他影响持续集成效果的 CI 反模式，请保持关注。

让开发自动化: 断言架构可靠性

通过主动构建过程掌控架构

您的软件架构和您所期望的一样吗？当架构落实到代码时，它并不总是 我们曾经互相讨论并预想的那个。在本期的 让开发自动化中，Paul Duvall 将演示如何通过使用 JUnit、JDepend 和 Ant 编写 有关测试来发现架构偏差，从而做到在发生问题之前主动发现问题。

我在曾经从事的很多软件开发项目中观察到，软件开发中一直存在这样一种现象：您实际拥有的架构往往与想象中的不同。

通过分析代码的度量报告，比如由 JDepend （参阅 Resources）工具生成的报告， 您可以有效地判定代码是否实现了确定的架构。有些团队对代码做反向设计，得到对应的 UML 图表，也能够达到上述效果， 还有一些团队甚至在编程时使用 IDE 生成相同的工件 —— 即实时反向设计。可是， 所有的这些方法都还是 反应式（*reactive*）的。 您必须手工审视并分析报告或图表，确定架构是否存在偏离，而有时这种 偏离可能很久之后才被发现。

设想每当某部分代码与 期望的架构有所违背时，您就得到一个提示 —— 比如一个 Ant 构建脚本失败 —— 如清单 1 所示：

清单 **1.** 违背架构导致构建失败

```
...
BUILD FAILED
...
build.xml:35 Test ArchitecturalRulesTest failed

Total time: 20 seconds
```

关于本系列

作为开发人员，我们的工作就是为终端用户实现过程自动化； 然而，很多开发人员却忽略了将自己的开发过程自动化的机会。 为此，我编写了 让开发自动化这个系列的文章，专门探讨软件开发过程自动化的实际应用，并教您 何时以及 如何成功地应用自动化。

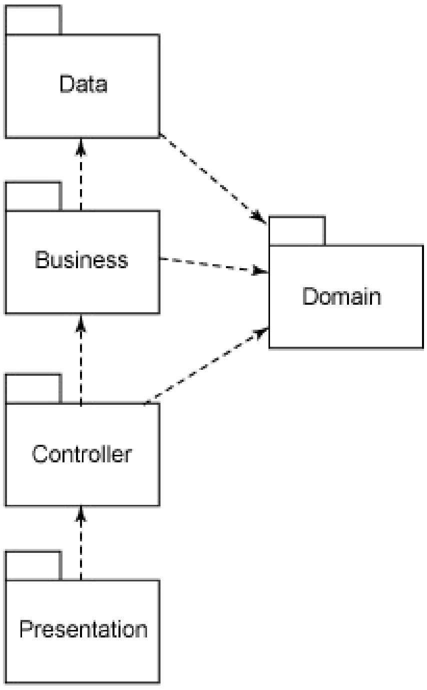
本文所提及的技术能够使您通过实现构建自动化主动分析软件架构。 除此之外，本文的示例还演示了如何基于规则触发构建过程失败，您可以使用 JDepend 的 API 和 JUnit 定义这些规则。

当然，最重要的是，通过构建自动化，您和您的团队能够在开发周期的 前期发现 源代码与确定架构之间的偏离。 这就是我所说的掌控架构！

反应式地设置开发阶段

图 1 说明了在构建 Web 应用时一种常见的架构模式。 `presentation` 层（代表一组相关的包）依赖于 `controller` 层， `controller` 层依赖于 `domain` 和 `business` 层，最后， `business` 层依赖于 `data` 和 `domain` 层。

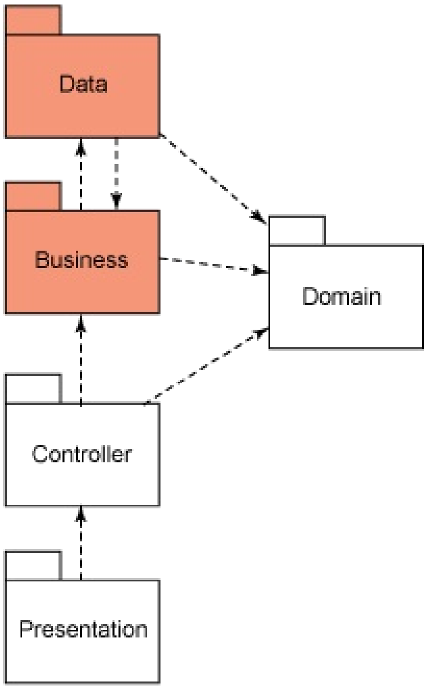
图 **1.** 典型 **web** 应用的一个架构分层图



至此，一切都很好，是吗？但是，我很确定您以前遇到类似的情形，那些常见的最佳实践规则会在日常的软件开发中被遗忘。事实上，这一点很容易（也很快）就会发生。

举例而言，图 2 阐明了对该示例架构的一个微小违背；在这个例子里， `data` 层至少调用一次 `business` 层：

图 **2. Data** 层正在调用 **Business** 层的一个对象，由此产生了架构违背



架构的微小变化产生的意外影响使代码的修改变得更加困难。实际上，现在对一个代码区域的修改 会要求其他很多区域的变动。举例而言， 如果您清除或改变 business 层中类的一些方法， 则可能需要从 data 层中清除某些引用。当更多的违背发生时， 修改代码就会更加困难。

若使用传统的监控技术，比如查看 JDepend 或 Macker 报告（参阅 参考资料）， 您能够多快地发现对期望设计的偏离呢？ 如果您和我一样，想快速地生产出能够工作的软件， 那么越快发现影响交付速度的问题越好， 难道您不这么认为吗？

使用简单的 JDepend

JDepend 是最方便的帮助评定架构违背的工具之一。经过几年的发展， 该开源工具能够很好地与 Ant 和 Maven 集成； 此外，它对大量的 Java™API 提供支持，具有更细化的交互性。但是， 如同我已经指出的，它生成的报告本质上是被动的。根据您实际运行（并查看）它们的频率， 架构违背可能直到很难矫正的时候才会被发觉。

传入耦合（Afferent coupling）与传出耦合（Efferent coupling）

JDepend 中，传入耦合表示一些包的数量，这些包依赖某个经过分析的包。比如说，如果您正在使用日志框架或一个象 Struts 一样的 Web 框架， 您会希望这些包具有高的传入耦合，因为整个代码库的很多包都依靠这些框架。 某包的传出耦合与传入耦合相反，是指某个经过分析的包所依赖的其他包的数量，也就是说，它具有的依赖包的数量。

断言架构

要主动判定架构变动是否恰当，实际上就是研究某特定包的耦合。事实上，通过对软件架构内关键包的传入和传出耦合进行监控， 以及观察预期值的偏离，您能轻松地发现错误的修改。

举例而言，在图 2 所示的修改中， `data` 层的新传出耦合现在大于 0，因为该层目前要 直接与 business 层通信。当然，这种耦合是通过 `data` 层中一个简单的 `import` （及对引用类的使用）而引入的。幸运的是，很容易通过 JUnit、JDepend 的优秀 API 发现此类问题，还有，构建时需要一些技巧。

事实证明，在构建（如 Ant 或 Maven）上下文中，您能够运行使用 JDepend API 的 JUnit 测试主动辨别耦合值的变化； 此外，如果这些变化不正确，您就可以使构建失败。 这就实现了主动性，不是吗？

第一步是创建一个 JUnit 测试并且对 JDepend 做相应配置，如清单 2 所示：

清单 **2.** 在 **JUnit** 中设置 **JDepend**

```
import junit.framework.TestCase;
import jdepend.framework.JavaPackage;
import jdepend.framework.JDepend;

public class ArchitecturalRulesTest extends TestCase {
  private static final String DIRECTORY_TO_ANALYZE =
      "C:/dev/project-sandbox/brewery/classes";
  private JDepend jdepend;
  private String dataLayer = "com.beer.business.data";
  private String businessLayer = "com.beer.business.service";
  private Collection dataLayerViolations = new ArrayList<String>();

  public ArchitecturalRulesTest(String name) {
    super(name);
  }

  protected void setUp()throws IOException {
    jdepend = new JDepend();
    jdepend.addDirectory(DIRECTORY_TO_ANALYZE);
    // Calling the businessLayer from the dataLayer is a violation
    dataLayerViolations.add(businessLayer);
  }
```

清单 2 很长，我们总结以下几个要点：

- 需要两个 JDepend 类： `jdepend.framework.JavaPackage` 和 `jdepend.framework.JDepend` 。
- 待分析的源类位置由 `DIRECTORY_TO_ANALYZE` 常量定义。JDepend 通过 调用 `JDepend.addDirectory` 扫描该目录，该操作通过一个 fixture 完成（即 `setUp()` 方法）。
- 要分析的包由 “Layer” `String` 定义。
- `dataLayerViolations``Collection` 添加了 `businessLayer``String` （表示一个包）来指明这是对期望架构的违背。

按照这四个要点，我已经有效地设置了 JDepend，以针对特定代码库发挥其魔力。 现在，我要设定一些精确的逻辑以说明耦合值的变化。

清单 3 中的 `testDataLayer()` 测试用例是架构断言的核心。 该方法可判定是否存在对 `dataLayer` 的任何违背 —— 如果 `isLayeringValid()` 方法（在下面的 清单 4中定义）返回 `false`，测试用例 就被认为失败，也意味着必然存在一处架构违背。

清单 **3.** 使用 **JDepend** 测试架构违背

```
public void testDataLayer() {
  if (!isLayeringValid(dataLayer, dataLayerViolations)) {
    fail("Dependency Constraint failed in Data Layer");
  }
}
```

清单 3 中测试用例所调用的方法如清单 4 所示：

清单 **4.** 循环查找每个包的传入耦合

`isLayeringValid()` 方法的目的确定 清单 2中 `DIRECTORY_TO_ANALYZE` 目录内所有包的传入耦合。 您可以在清单 4 底部看到，该方法遵守 `isEfferentsValid()` 方法，如清单 5 所示。

这里，如果 `isEfferentsValid()` 方法发现某个包不符合指定的包依赖关系（由于从一个包到另一个包的传出耦合大于 0），则使用 清单 2中的 `dataLayerViolations` 集合将该包标记为一个架构违背。这将间接导致 `testDataLayer()` 测试用例（如 清单 3所示）失败。

清单 **5.** 判定包依赖关系违背

```
private boolean isLayeringValid(String layer, Collection rules) {
  boolean rulesCorrect = true;
  Collection packages = jdepend.analyze();
  Iterator itor = packages.iterator();
  JavaPackage jPackage = null;
  String analyzedPackageName = null;
  while (itor.hasNext()) {
    jPackage = (JavaPackage) itor.next();
    analyzedPackageName = jPackage.getName();
    Iterator afferentItor = jPackage.getAfferents().iterator();
    String afferentPackageName = null;
    while (afferentItor.hasNext()) {
      JavaPackage afferentPackage = (JavaPackage) afferentItor.next();
      afferentPackageName = afferentPackage.getName();
    }
    rulesCorrect = isEfferentsValid
        (layer, rules, rulesCorrect, jPackage, analyzedPackageName);
  }
  return rulesCorrect;
}
```

正如您所看到的，清单 2 到 5 实际上都是扫描一系列包以确定耦合变化； 如果耦合发生了变化，失败条件被触发，因此 JUnit 报告测试失败。 要让我说的话，这真是令人印象深刻！

别忘了自动运行测试

一旦您结合使用 JUnit 和 JDepend 编写好基于约束的测试后， 您就能够用诸如 Ant 或 Maven 这样的工具把它作为构建过程的一部分运行。 举例而言，清单 6 阐述了用 Ant 运行一系列此类测试。 `test.dependency.dir` 属性 映射到 root/src/test/java/dependency 目录，其中包含了一些神奇的架构验证程序。

清单 **6.** 运行依赖性约束测试的 **Ant** 脚本

```
<target name="run-tests" depends="compile-tests">
 <mkdir dir="${logs.junit.dir}" />
 <junit fork="yes" haltonfailure="true"dir="${basedir}" printsummary="yes">
   <classpath refid="test.class.path" />
   <classpath refid="project.class.path"/>
   <formatter type="plain" usefile="true" />
   <formatter type="xml" usefile="true" />
   <batchtest fork="yes" todir="${logs.junit.dir}">
     <fileset dir="${test.dependency.dir}">
       <patternset refid="test.sources.pattern"/>
     </fileset>
   </batchtest>
 </junit>
</target>
```

要使 JUnit 测试成功执行，JDepend JAR 必须出现在 Ant 的类路径中。 `haltonfailure` 属性被设为 true，以便让构建过程在测试失败时停止。

阈值驱动的架构

我已经指出，使用被动的方法维持架构需要付出大量的努力， 另外，我希望我已经使您相信，开发过程中很容易发生架构违背。 通过将架构测试作为构建过程的一部分执行， 您能够使这种检查自动化并且能够重复执行。图 3 显示了在运行 Ant 后显示构建失败，这样不是很好吗？我甚至根本不需要再去看 JDepend 报告了。

图 **3.** 架构违背引起的构建失败

这种主动监控的优势在于，你可以在发现架构分层问题后马上解决它。 问题解决得越迅速，越有助于降低风险 —— 更不用提代价了。 本质上，您的团队不会因此收到干扰，并能够继续工作，实现快速发布可用软件的目标。

针对架构的自动化

JDepend 还有哪些魔力？

存在多种方法通过 JDepend 添加主动检查。实际上，JDepend 建议使用其 `DependencyConstraint` 类。尽管使用 DependencyConstraint 非常简单，但我还是不选择它，因为 它只具有使用 API 执行架构规则这么一种途径，而且不能可靠地根据我的需求工作。还有其他一些工具支持包依赖关系遵从性； 可参阅 参考资料以了解更多细节。

现在您能够用自己的构建过程主动发现与期望架构的设计违背了。 此外，我已向您展示了几个可能的示例之一 —— 您一定能够获得创造性的方法并分析类似 *Instability*包这样的度量，从而便于判定架构的整体健壮性。

我所介绍的这个方法是一种简单的方式，可以减少 为判定架构遵从性而不断反向设计代码并分析图表的需求。 如果您在使用持续集成系统（Continuous Integration system）， 您可将这些测试用作一个安全网络，确保检查版本控制系统的代码传递这些架构规则 —— 每当进行了一次改动时。如果您改变了架构， 只要改变您的 JUnit 测试规则，就可确保 您的团队遵守了项目标准。这就是我所称的使用主动方式断言架构可靠性。

让开发自动化: 持续测试

对代码基址的每一次更改都运行自动化测试

准备好开始在您的开发人员测试活动中大获全胜吗？在本期的 让开发自动化 中，开发自动化专家 Paul Duvall 介绍了几种自动化的开发人员测试，每一次改变源代码都能够运行这些测试。Paul 提供了 Selenium、DbUnit 和 JUnitPerf 测试的例子，即，如果经常 运行这些测试可以帮助您尽早发现应用程序的问题。

在像 Eclipse 那样的 IDE 中或者比如在 Ant 构建脚本中运行单元测试是确保应用程序质量的一个很好的开始；然而，版本控制库（如 Subversion）中的源代码一改变，在单独无变动的构建机上运行单元测试就有助于检验开发生命周期中的问题。而且，运行各种类型的开发人员测试，如组件测试、功能测试和性能测试，能够在开发生命周期中更早地 将问题显示出来。

关于本系列

作为开发人员，我们的工作就是为终端用户实现过程自动化；然而，很多开发人员却忽略了将自己的开发过程自动化的机会。为此，我编写了 让开发自动化 这个系列的文章，专门探讨软件开发过程自动化的实际应用，并教您何时 以及如何 成功地应用自动化。

通常在持续集成（CI）环境中运行的开发人员测试有效地扮演着代码质量聚光灯的角色。这是因为如果能有效地编写这些测试，则几乎能够在问题（如缺陷）产生之时就将其发现。不经常运行测试通常就不怎么有效，因为从产生缺陷到发现该缺陷的时间相隔很长，但持续地（即，每一次代码改变时）运行测试能确保快速发现无意识的行为。

本文涵盖下列内容：

- 通过 Ant 运行 JUnit 测试
- 使用 JUnit 和 DbUnit 执行更长时间的运行组件测试
- 使用 JUnitPerf 确定哪些方法花费时间过久而执行失败
- 用 Selenium 运行基于 Web 的功能测试
- 用 Cobertura 访问代码覆盖率
- 用 CruiseControl 进行持续测试

我提供一个关于不同类型开发人员测试的概览，和一些可以添加到构建过程并使用 Continuous Integration 系统持续运行的例子。

按 JUnit 进行单元测试

有人称之为单元测试，有人称之为组件测试

经常提到的单元测试其实更像是一个组件级别的测试。组件测试通常验证不止一个类，并且依赖于一些东西，如数据库或其他重量级系统，如文件系统。但是更重要的是，在测试的基础上进行测试，组件测试比单元测试运行时间更长。

有时，我听到开发人员将开发人员测试这一术语与简单的单元测试相混淆；然而，我发现将单元测试这一术语提练得更加明确很有帮助。对我来说，单元测试是快速运行的 测试，通常测试没有大的外部依赖项（如数据库）的单独的类。例如，清单 1 定义了一个单元测试，该测试使用 JUnit 来验证一个叫做 `BeerDaoStub` 的存根数据类。针对并未真正连接到数据库的接口的测试技术是一种验证业务功能的方法，使用该方法不会导致花费昂贵的设置成本。另外，这样做可使测试保持为一个真正的单元测试。

清单 **1.** 一个简单的单元测试

```
public void setUp() {
  beerService = new BeerDaoStub();
}

public void testUnitGetBeer() {
  Collection beers = beerService.findAll();
  assertTrue(beers != null && beers.size() > 0);
}
```

一旦编写了一些单元测试，就可以一直通过 IDE 运行这些测试，但您也想要将这些测试作为构建过程的一部分来运行。确保该测试通过构建过程成功运行意味着也能从 CI 构建的上下文中启动这些相同的测试。

清单 2 是一个 Ant 脚本片段，介绍了执行一批单元测试的 `junit` 任务。这项任务与 JUnit 一起运作，其妙处在于：定义过的所有测试现在都能自动运行并且如果其中任何一个测试失败，则构建也将失败 —— 通过使用 `haltonfailure` 属性实现。

清单 **2.** 在 **Ant** 中运行单元测试

```
<junit fork="yes" haltonfailure="true" dir="${basedir}" printsummary="yes">
  <classpath refid="test.class.path" />
  <classpath refid="project.class.path"/>
  <formatter type="plain" usefile="true" />
  <formatter type="xml" usefile="true" />
  <batchtest fork="yes" todir="${logs.junit.dir}">
    <fileset dir="${test.unit.dir}">
      <patternset refid="test.sources.pattern"/>
    </fileset>
  </batchtest>
</junit>
```

注意： `test.unit.dir` 指定测试的位置。这是将这些测试（在本例中为单元测试）和其他测试隔离起来的有效方法。通过利用这项技术，可以通过定义另外的 Ant 目标来先运行较快的测试，接着运行较慢的测试（如组件测试、功能测试和系统测试）。

集合组件测试

由于单元测试执行得相当快，很容易将它们作为构建的一部分经常运行。但这些测试并未达到一个高的代码覆盖率 —— 其隔离的本质决定了它们只测试一部分功能。编写具有更多代码（从而可实现更多功能）的测试通常要以附属框架的形式执行更多的调查工作。一旦开始使用这些帮助框架来编写测试，这些测试就开始成为更高级别的测试，我把它们归类为组件测试。

组件测试是基本的测试，这些测试将验证不止一个类，且通常依赖于外部依赖项，如数据库。组件测试的编写方式和单元测试大体一致，只是前者并非通过模拟或存根类来强制隔离，实现这些测试可谓勉为其难，但可以利用框架来便利对外部依赖项的使用。例如，我通常使用 DbUnit 框架来帮助管理数据库，以便组件测试可验证依赖数据库数据的代码功能。

用 **DbUnit** 控制数据库状态

DbUnit 是一个框架，它使针对数据库的测试过程变得更加简单。它提供了一个标准 XML 格式，用于定义一些测试数据，以便从数据库中选择、更新、插入和删除数据。请牢记，DbUnit 并没有替换数据库；它只是提供了一种更加有效的机制来处理测试数据。您可以用 DbUnit 来编写依赖于特定数据的测试，DbUnit 保证该数据位于底层的数据库中。

可以在 JUnit 中可编程地使用 DbUnit，或者可以将它作为构建过程的一部分使用。该框架带有一个 Ant 任务，该任务提供了一种使用 XML 文件来操作、导出或比较数据库中数据的方法。例如，清单 3 演示了 `dbunit` 任务，在本文的例子中，该任务将测试数据插入到目标数据库中，然后在运行完所有组件测试后删除数据：

清单 **3.** 在 **Ant** 中运行组件测试

```
<target name="component-tests">
  <mkdir dir="${logs.junit.dir}" />
  <taskdef name="dbunit"
    classname="org.dbunit.ant.DbUnitTask"/>
    <dbunit driver="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/brewery"
      userid="${db.username.system}"
      classpathref="db.lib.path"
      password="${db.password.system}">
      <operation type="INSERT"
        src="seedFile.xml"/>
    </dbunit>
    <junit fork="yes" haltonfailure="false"
      failureproperty="tests.failed"
      haltonerror="true" dir="${basedir}"
      printsummary="yes">
      <classpath refid="test.class.path" />
      <classpath refid="project.class.path"/>
      <formatter type="plain" usefile="true" />
      <formatter type="xml" usefile="true" />
      <batchtest fork="yes" todir="${logs.junit.dir}">
        <fileset dir="${test.component.dir}">
          <patternset refid="test.sources.pattern"/>
        </fileset>
      </batchtest>
    </junit>
    <mkdir dir="${reports.junit.dir}" />
    <junitreport todir="${reports.junit.dir}">
      <fileset dir="${logs.junit.dir}">
        <include name="TEST-*.xml" />
        <include name="TEST-*.txt" />
      </fileset>
      <report format="frames" todir="${reports.junit.dir}" />
    </junitreport>
    <dbunit driver="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/brewery"
        classpathref="db.lib.path"
        userid="${db.username.system}"
        password="${db.password.system}">
      <operation type="DELETE"
          src="seedFile.xml"/>
    </dbunit>
</target>
```

正如清单 3 所示，现在组件测试可在执行期间依赖驻留在数据库中的特定数据。另外，由于在所有测试成功执行后删除了所有的数据，因而此过程现在可重复执行。

在数据库中播种

可以将 `dbunit` 任务的 `INSERT` 和 `DELETE` 操作类型和一个种子文件起使用，该文件包含表示数据库表和相关行的 XML 元素。例如，清单 4 是清单 3 中引用的 `seedFile.xml` 文件的内容。每个 `BEER` 元素表示一个也叫 `BEER` 的数据库表，`BEER` 元素的每个属性和其值都映射至相应的数据库列名称和值。

清单 **4. DbUnit** 种子文件

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <BEER id='6'
            beer_name='Guinness Extra Stout'
            brewer='St.James Brewery'
            date_received='2007-02-01' />
  <BEER id='7'
            beer_name='Smuttynose Robust Porter'
            brewer='Smuttynose Brewery'
            date_received='2007-02-01' />
  <BEER id='8'
            beer_name='Wolavers pale ale'
            brewer='Wolaver Brewery'
            date_received='2007-02-01' />
</dataset>
```

您也许已经从清单 3 中注意到，可以在不同的操作中重用 DbUnit 的种子文件。在本文的例子中，将在运行组件测试前使用清单 4 中的文件在数据库中播种，然后使用相同的文件指示测试完成时从数据库中删除哪些数据。

参与性能测试

开发人员完成编码后，常常要经过很长时间才执行性能测试，而事实通常是可以在开发周期中更早的时候发现（并且解决）性能问题。幸运地是，有一种方法可解决此问题：持续测试或更具体地、持续地运行 JUnitPerf 测试。

对性能测试来说 JUnitPerf 是完美的

JUnitPerf 是一个同 JUnit 协调工作的框架，该框架在一个预定的时间限制内执行测试用例：如果一个测试中的方法所用的时间比预期的阈值长，则认为该测试是失败的。通过将性能测试集成到自动化构建中，您能有效地监控应用程序的性能甚至能在出现性能问题时使构建失败。

但我倾向于将 JUnitPerf 用作一种发现早期性能问题的简单方法，而不是将其作为一种机制来衡量执行时间；像 profilers 这样的工具更善于提供此类衡量。在本质上，可以认为 JUnitPerf 是一个早期的警告系统。

在清单 5 中，我定义了一个 JUnit 测试，该测试使用 JUnitPef 来验证 `BeerServicePerformanceTest` 测试类中的 `testLongRunningMethod` 测试的执行时间。如果执行该测试方法所花的时间多于 1000 毫秒，则测试失败。

清单 **5.** 使用 **JUnitPerf** 的基于性能的测试

```
package com.beer.business.service;
import com.clarkware.junitperf.*;
import junit.framework.Test;

public class ExampleTimedTest {
  public static Test suite() {
    long maxElapsedTime = 1000;
    Test testCase = new BeerServicePerformanceTest("testLongRunningMethod");
    Test timedTest = new TimedTest(testCase, maxElapsedTime);
    return timedTest;
  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }

}
```

使用精确计时作为方法执行时间的标准时要小心；测试的建立和销毁时间包含在整个执行时间中。此外，在早期的性能测试中，精确测定执行速度在更大程度上是一门艺术而不是科学。

使用 **Selenium** 进行功能测试

可随意编写所有需要的单元测试和组件测试，但如果要编写一个提供某种类型的用户界面的应用程序（例如 Web 应用程序），则需要测试表示层。以 Web 应用程序为例，需要验证用户场景的导航，另外还要验证场景的功能是正常的。尽管如此，直到最近，这类测试都常被证明是一个负担，需要购买工具来促进开发周期晚期的测试。此外，这些工具几乎不能适合构建过程，即使测试构建得足够早也是如此。

深入 **Selenium**

但近几年来，一些着眼于功能测试的开放源码工具脱颖而出；而且，能轻易地在开发生命周期的早期使用这些工具。工具如 Selenium 和 Watir 都是开放源码的；另外，它们构建时考虑到了开发人员。除了用各种语言（例如 Java 编程和 Python）编程定义 Selenium 测试之外，Selenium 也提供了一种易于学习的表格驱动格式，此格式也能被非技术类型使用。

Selenium 框架使用 JavaScript 来执行基于 Web 的接受测试，该测试打开一个浏览器并运行表格驱动测试。例如，清单 6 展示了一个表示简单的 Selenium 测试的 HTML 表。该测试的多个步骤打开一个 Web 应用程序，然后使用有效的用户名和密码执行登录。测试结果生成到一个 HTML 表中，在 Selenium 运行完所有的测试后，能查看该表。

清单 **6.** 使用 **Selenium** 的功能测试

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>MyTest</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
</thead><tbody>
<tr>
  <td>open</td>
  <td>/beer/</td>
  <td></td>
</tr>
<tr>
  <td>type</td>
  <td>username</td>
  <td>admin</td>
</tr>
<tr>
  <td>type</td>
  <td>password</td>
  <td>password</td>
</tr>
<tr>
  <td>clickAndWait</td>
  <td>//input[@value='Login']</td>
  <td></td>
</tr>
<tr>
  <td>verifyTextPresent</td>
  <td>Logged in as admin</td>
  <td></td>
</tr>
</tbody></table>
</body>
</html>
```

使用清单 6 中基于表格的格式，可以定义多个接受测试。也可以将测试分组成套，一次执行一整套测试。

使用 Ant 驱动 Selenium

Selenium 的伟大之处在于它是在考虑了 CI 的基础上从头创建的，因为你能在像 Ant 那样的构建工具中运行 Selenium。此外，由于框架设计者的高瞻远瞩，如果任何 Selenium 接受测试失败，您也可以让整个构建失败。例如，清单 7 展示了一个 Ant 任务，该任务使用 Selenium 远程控制服务器在一个 Web 应用程序中执行一系列表格驱动测试：

清单 7. 使用 Ant 运行 Selenium

```
<?xml version="1.0" encoding="iso-8859-1"?>
<project name="functional-tests" default="run-selenium-tests" basedir=".">
  <property file="${basedir}/selenium.properties"/>
  <import file="${basedir}/common-environment.xml"/>
  <property name="acceptance.test.lib.dir"
              value="${functional.test.dir}" />
  <property name="firefox" value="*firefox" />
  <property name="base.url"
              value="http://${web.host.name}:${web.port}" />
  <property name="acceptance.test.list.dir"
              value="${functional.test.dir}" />
  <property name="acceptance.test.report.dir"
              value="${functional.test.dir}" />
  <target name="run-selenium-tests">
    <mkdir dir="${reports.dir}" />
    <java jar="${acceptance.test.lib.dir}/selenium-server.jar"
          fork="true">
      <arg line="-htmlSuite "${firefox}""/>
      <arg line=""${base.url}""/>
      <arg line=""${acceptance.test.list.dir}/${test.suite}""/>
      <arg line=""${reports.dir}/index.html""/>
      <arg line="-timeout ${timeout}"/>
    </java>
  </target>
  <target name="stop-server">
    <get taskname="selenium-shutdown"
         src="http://${web.host.name}:
           ${selenium.rc.port}/selenium-server/driver/?cmd=shutDown"
      dest="result.txt" ignoreerrors="true" />
     <echo taskname="selenium-shutdown"
            message="Errors during shutdown are expected" />
  </target>
</project>
```

执行 Selenium 测试时，当框架打开 Web 浏览器、闪电般执行测试，然后关闭该浏览器并生成 HTML 报告时，不要被吓到。这是一种在开发生命周期的早期更快更容易地发现问题的方法（此时它们更易处理）。

使用 Cobertura 报告代码覆盖率

是否达到 100% 就是问题所在

运行像 Cobertura 或者 Emma 这样的工具时，记住以下方面很重要：在一个特殊的方法中实现 100% 的行覆盖并不意味着该方法没有缺陷或者它已被完全测试。例如，如果您编写了一个针对 `if` 语句的测试，该测试包含逻辑 `And`，而测试针对的是表达式的左侧部分，则像 Cobertura 这样的工具将报告 100% 行覆盖，但是实际上，您仅执行了该语句的 50%；因此仅完成了 50% 的分支覆盖。

现在已经编写了一些测试，如何确定所有这些测试执行什么 呢？幸运的是，此问题可由像 Cobertura 这样的代码覆盖工具来解答。代码覆盖工具可报告测试覆盖率 —— 以行覆盖或分支覆盖形式表示 —— 它表示测试运行时所涉及的代码量。

清单 8 展示了一个 Ant 脚本。该脚本使用 Cobertura 生成一份关于代码覆盖率的 HTML 报告，代码覆盖率通过运行一系列 JUnit 测试获得：

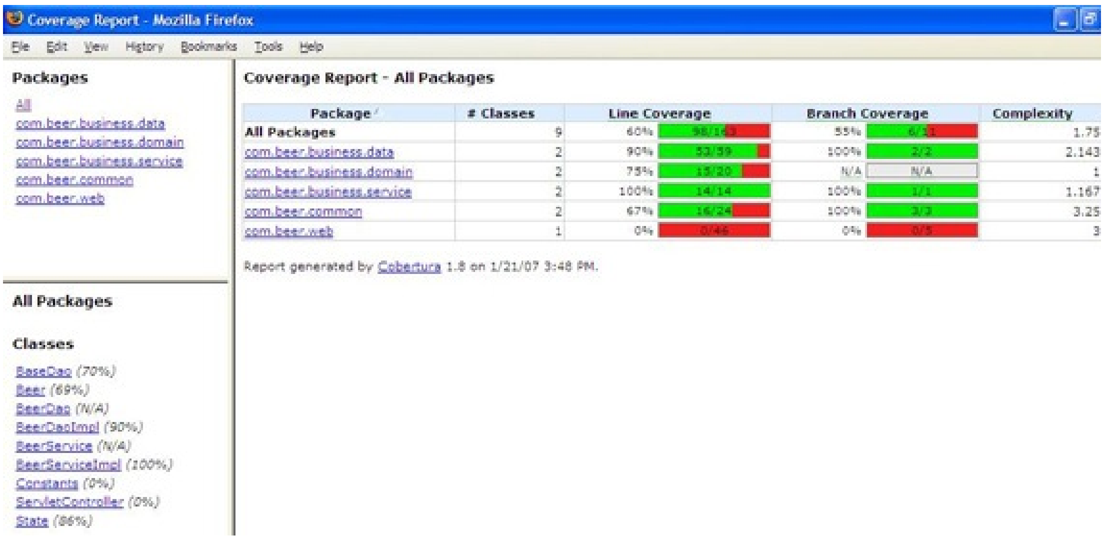
清单 8. 使用 Ant 和 Cobertura 报告代码覆盖率

```
<target name="instrument-classes">
  <mkdir dir="${instrumented.dir}" />
  <delete file="cobertura.ser" />
  <cobertura-instrument todir="${instrumented.dir}">
    <ignore regex="org.apache.log4j.*" />
    <fileset dir="${classes.dir}">
      <include name="**/*.class" />
      <exclude name="**/*Test.class" />
    </fileset>
  </cobertura-instrument>
</target>

<target name="run-instrumented-tests" depends="instrument-classes">
  <mkdir dir="${logs.junit.dir}" />
  <junit fork="yes" haltonfailure="true" dir="${basedir}" printsummary="yes">
    <sysproperty key="net.sourceforge.cobertura.datafile" file="cobertura.ser" />
    <classpath location="${instrumented.dir}" />
    <classpath location="${classes.dir}" />
    <classpath refid="test.class.path" />
    <classpath refid="project.class.path"/>
    <formatter type="plain" usefile="true" />
    <formatter type="xml" usefile="true" />
    <batchtest fork="yes" todir="${logs.junit.dir}">
      <fileset dir="${test.component.dir}">
        <patternset refid="test.sources.pattern"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```

Cobertura 产生了一个如图 1 中所示的 HTML 报告。请注意行覆盖和分支覆盖的百分比是以包计算的。可单击每一个包，获得类级别的行百分比和路径百分比，甚至能看到执行的源代码行和它们执行的次数。

图 1. 使用 Cobertura 和 Ant 生成 HTML 报告



需要多高的代码覆盖率?

理想情况下，您可能想针对每个路径执行一次测试。也就是说，如果整个代码基址有 20,000 的循环复杂度的话，则需要 20,000 次测试。我从未遇到过具有 100% 的路径覆盖的项目，不过我曾见过具有近 100% 的行覆盖的团队。

已经介绍了多种类型的测试，甚至介绍了如何测量这些测试的覆盖率 —— 但是如何确保以正常的间隔执行 这些测试呢？恰好，这正是 CI 服务器（如 CruiseControl）大显身手的地方，接下来对它进行介绍。

持续运行测试

一旦将这些各式各样的开发人员测试类型合并到一个构建过程中时，可以将这些测试中的一些（或者全部）作为 CI 过程的一部分运行。例如，清单 9 是 CruiseControl 的 `config.xml` 文件的一个片段，我在其中定义了一些东西。首先，我让 CruiseControl 每两分钟监控一次 Subversion 库中的改变。如果发现任何改变，则 CruiseControl 将启动一个叫做 `build-${project.name}.xml` 的委托 构建脚本（通常，此脚本用 Ant 编写）。该委托构建脚本调用项目的构建脚本，后者执行编译并运行测试。

我也定义了一些逻辑，将所有不同类型的测试结果合并到一个 CruiseControl 日志文件中。而且，我还利用 CruiseControl 的功能将不同工具生成的报告链接（使用 `artifactspublisher` 标签）到 Build Artifacts 链接中，Build Artifacts 可以从 CruiseControl 的显示板应用程序中获得。

清单 **9.** 使用 **CruiseControl** 的 **CI**

```
...
<modificationset quietperiod="30">
  <svn RepositoryLocation="http://your-domain.com/trunk/brewery"
    username="bfranklin"
    password="G0Fly@Kite"/>
</modificationset>
<schedule interval="120">
  <ant anthome="apache-ant-1.6.5" buildfile="build-${project.name}.xml"/>
</schedule>
<log dir="logs/${project.name}">
  <merge dir="projects/${project.name}/_reports/unit"/>
  <merge dir="projects/${project.name}/_reports/component"/>
  <merge dir="projects/${project.name}/_reports/performance"/>
  <merge dir="projects/${project.name}/_reports/functional"/>
  <merge dir="projects/${project.name}/_reports/coverage"/>
</log>
<publishers>
  <artifactspublisher
    dir="projects/${project.name}/_reports/"
    dest="projects/artifacts/${project.name}"/>
</publishers>
...
```

在将每个源变更应用到版本控制库中时，不必运行每个定义的测试。例如，可以设置 CI 系统执行构建（通常称作提交构建），该构建只在代码签入时运行单元测试。可以为提交构建补充一些更重量级的构建，例如像运行组件测试、功能测试、性能测试以及甚至执行代码检查的构建（请参阅 参考资料）。这些构建可以以更低的频率运行（如一天一次）。您也可以选择在提交构建之后立即运行这些测试和检查。

调用所有测试

持续测试包括了广度和频度。通过授权执行不同类型的测试，可获得更大范围的覆盖和信任。此外，通过持续运行这些测试，几乎能在问题产生就发现它们。

仅仅进行单元测试（至少我所定义的单元测试），并不能使你在项目上走得很远。取得更高的代码覆盖率并且增加团队的信心，需要通力合作并执行自动化组件测试、性能测试和功能测试。此外，使用框架和像 JUnit、Selenium 以及 Cobertura 这样的工具能轻松实现构建自动化，这也意味着在 CI 系统的帮助下，能够在每次将变更提交到版本控制库中时，有效地执行测试套件。这肯定是一种万无一失的提高平均成功率的方法，您不这么认为吗？

让开发自动化: 用 Eclipse 插件提高代码质量

在 Eclipse 中使用 5 个有用的插件来自动化代码质量分析

如果能在构建代码前发现代码中潜在的问题会怎么样呢？很有趣的是，Eclipse 插件中就有这样的工具，比如 JDepend 和 CheckStyle，它们能帮您在软件问题暴露前发现这些问题。在 让开发自动化 的本期文章中，自动化专家 Paul Duvall 将带来一些关于 Eclipse 插件的例子，您可以安装、配置和使用这些静态分析插件，以便在开发生命周期的早期预防问题。

关于本系列

作为一名开发人员，我们的工作就是为终端用户将过程自动化；然而，我们当中有很多人却忽视了将我们自己的开发过程自动化的机会。为此，我编写了让开发自动化 这个系列的文章，专门探索软件开发过程自动化的实际应用，并教您何时 以及如何 成功地应用自动化。

开发软件时，我的主要目标之一是：要么防止将缺陷引入代码库，要么限制缺陷的生存期；换言之，要尽早找到缺陷。很显然，越是了解如何编写更好的代码以及如何有效测试软件，就越能及早地捕捉到缺陷。我也很想要一张能发现潜在缺陷的安全之网。

在本系列 八月份 的那期文章中，我得出了这样的结论：将检验工具集成到构建过程（例如，使用 Ant 或 Maven）中，能够建立起一种寻找潜在缺陷的方法。尽管这种方法使一致性成为可能并超越了 IDE，但它也有一点反作用。必须在本地构建软件或等待 Continuous Integration 构建的运行。如果使用 Eclipse 插件，就可以在通过 Continuous Integration 构建或集成前 发现一些这样的冲突。这就促成了我称为渐进编程 的编程方式，在这种方式下，允许在编码过程中进行一定程度的质量检验 —— 再也不能比这个更早了！

本文涵盖了我所认为的“五大”代码分析领域：

- 编码标准
- 代码重复
- 代码覆盖率
- 依赖项分析
- 复杂度监控

可以用接下来的几个灵活的 Eclipse 插件来揭示这些分析领域：

- CheckStyle：用于编码标准
- PMD 的 CPD：帮助发现代码重复
- Coverlipse：测量代码覆盖率
- JDepend：提供依赖项分析
- Eclipse Metric 插件：有效地查出复杂度

Eclipse 不是您的构建系统

使用 Eclipse 插件与您将这些检验工具用于构建过程并不矛盾。事实上，您想要确保的是：下列使用 Eclipse 插件的规则就是应用到构建过程中的规则。

安装 Eclipse 插件

安装 Eclipse 插件再简单不过了，只需要几个步骤。在开始之前，最好把该插件下载站点的 URL 准备好。表 1 是本文用到的插件的列表：

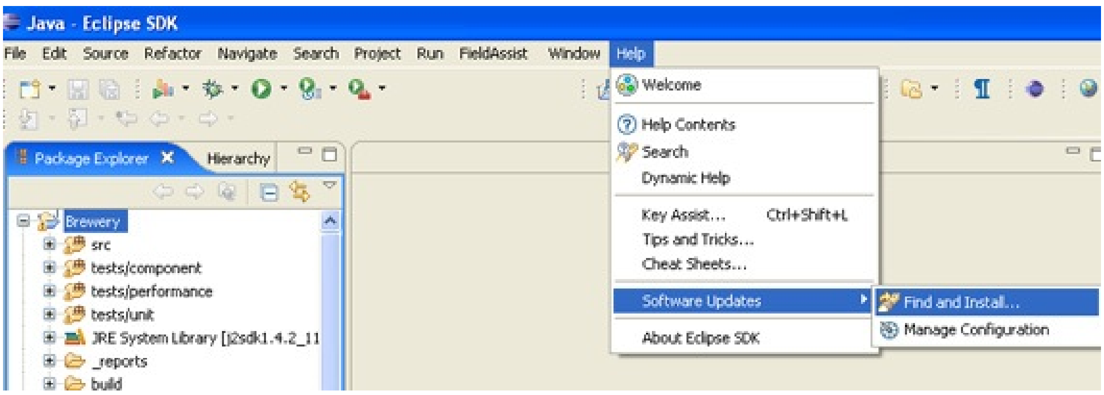
表 **1.** 代码改进插件和相应的下载站点 **URL**

工具	目的	**Eclipse** 插件的 **URL**
CheckStyle	编码标准分析	http://eclipse-cs.sourceforge.net/update/
Coverlipse	测试代码覆盖率	http://coverlipse.sf.net/update
CPD	复制/粘贴检验	http://pmd.sourceforge.net/eclipse/
JDepend	包依赖项分析	http://andrei.gmxhome.de/eclipse/
Metrics	复杂度监控	http://metrics.sourceforge.net/update

知道了这些有用插件的下载地址后，安装插件就是一个极简单的过程。启动 Eclipse，然后遵循下列步骤：

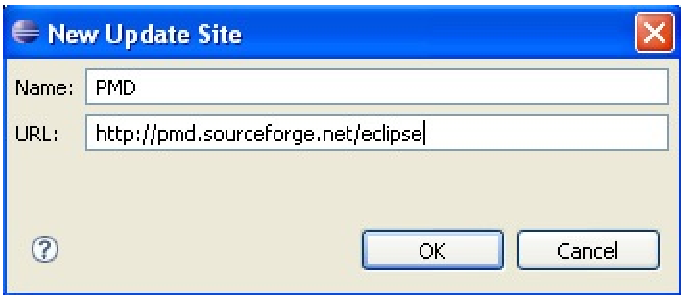
1. 选择 **Help | Software Updates | Find and Install**，如图 1 所示：

   图 **1.** 寻找并安装 **Eclipse** 插件

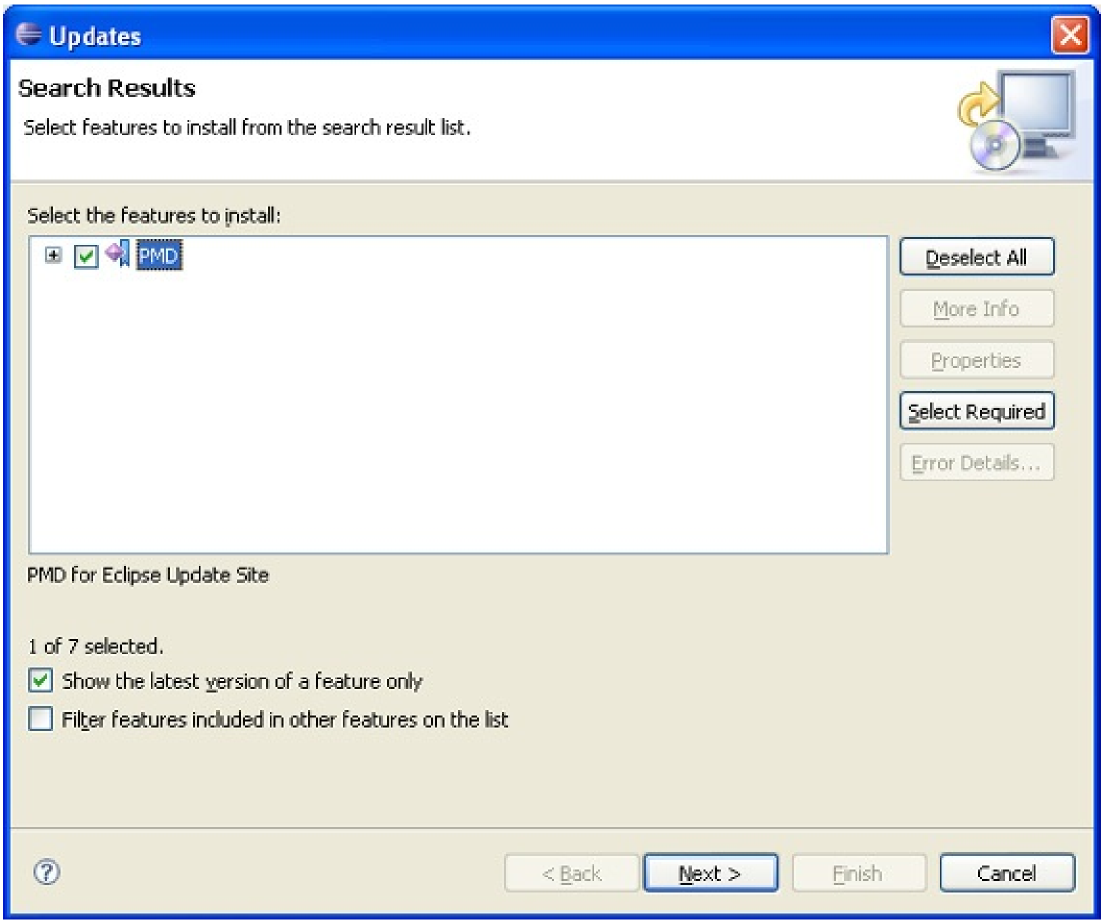
2. 选择 **Search for new features to install** 单选按钮,单击 **Next**。
3. 单击 **New Remote Site**，输入要安装的插件名和 URL（参见图 2），单击 **OK**，然后单击 **Finish** 来显示 Eclipse 更新管理器。

   图 **2.** 配置新的远程站点



4. 在 Eclipse 更新管理器中，有一个查看插件各方面特性的选项。我通常选择顶级项，如图 3 所示。选择您需要的选项并单击 **Finish**。Eclipse 现在安装该插件。您需要重启 Eclipse 实例。

   图 **3.** 安装 **Eclipse** 插件



请遵循上述这些步骤来安装其他的 Eclipse 插件；只需改变插件名和相应的下载位置即可。

用 CheckStyle 校正标准

代码库的可维护性直接影响着软件的整个成本。另外，不佳的可维护性还会让开发人员十分头痛（进而导致开发人员的缺乏）—— 代码越容易修改，就越容易添加新的产品特性。像 CheckStyle 这样的工具可以协助寻找那些可影响到可维护性、与编码标准相冲突的地方，比

方说，过大的类、太长的方法和未使用的变量等等。

有关 PMD

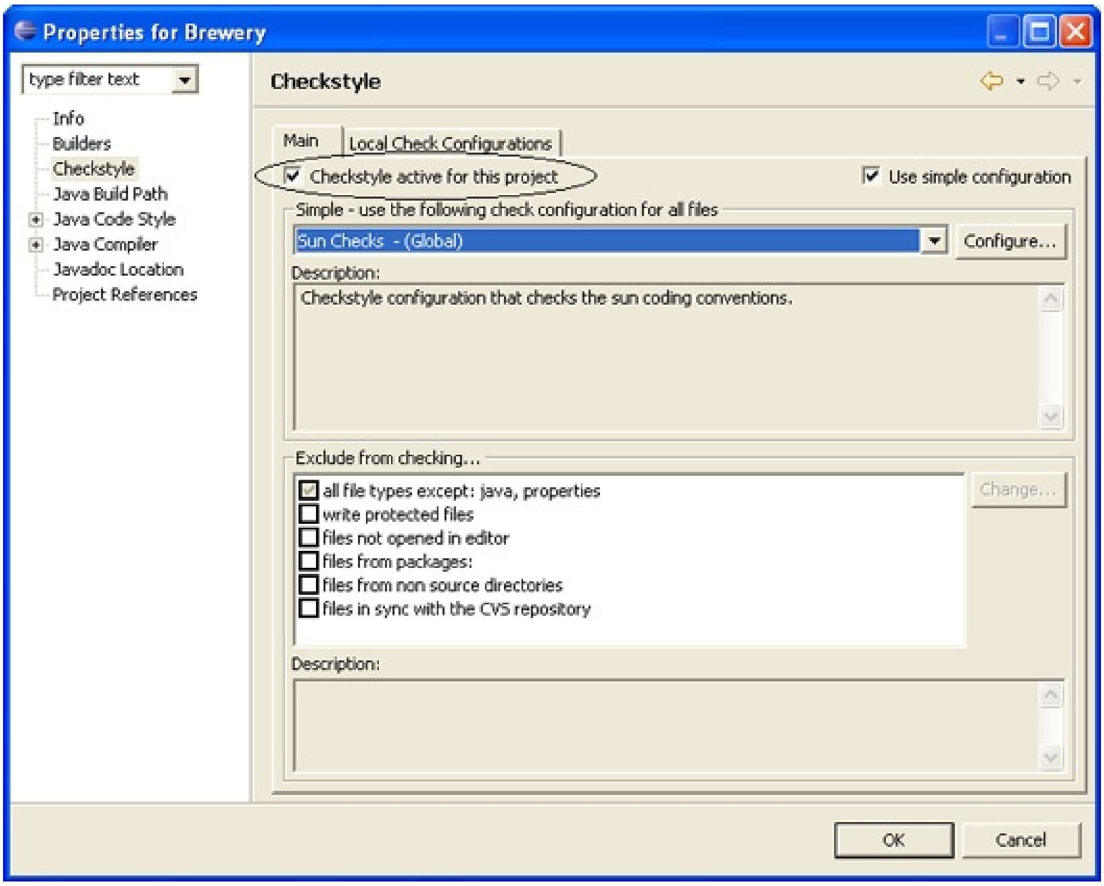
另一个叫做 PMD 的开源工具提供的功能和 CheckStyle 类似。我偏爱 CheckStyle，但 PMD 也有很多执着的追随者，所以我建议您了解一下这个工具，毕竟它也颇受一些人的青睐。

使用 Eclipse 的 CheckStyle 插件的好处是能够在编码过程中了解到源代码上下文的各种编码冲突，让开发人员更可能在签入该代码前真正处理好这些冲突。您也几乎可以把 CheckStyle 插件视作一个连续的代码复查工具！

安装 CheckStyle 插件并做如下配置（参见图 4）：

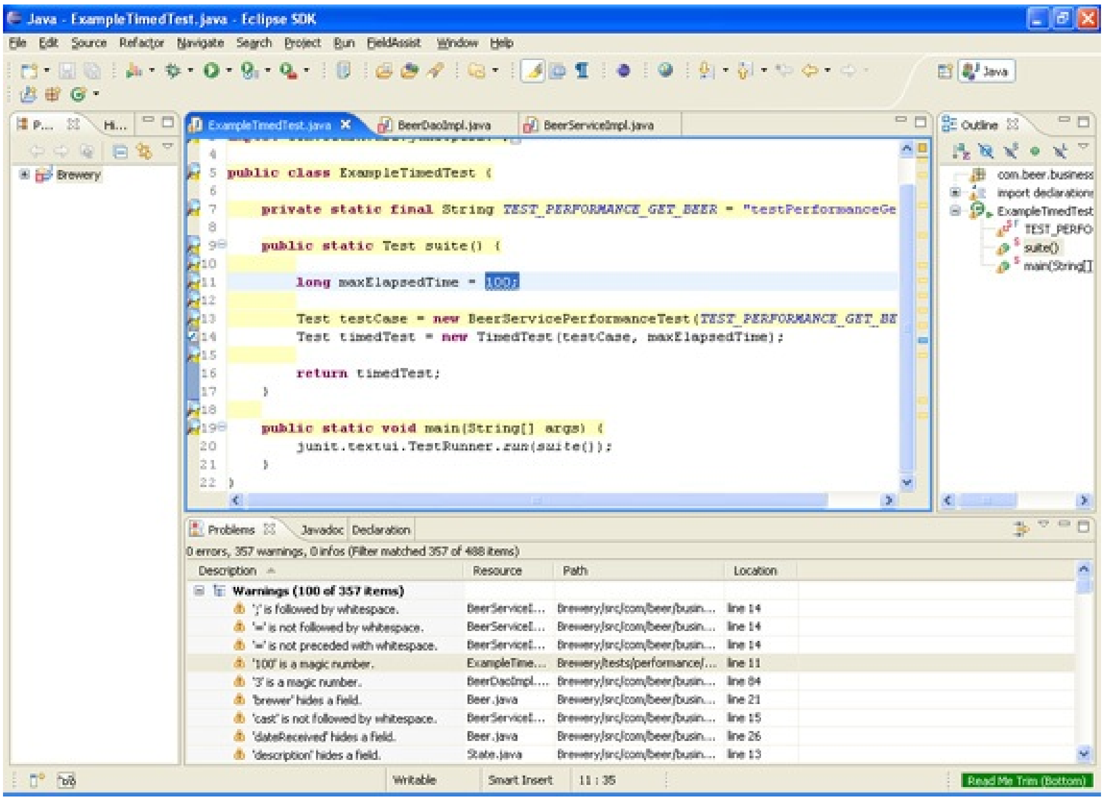
1. 选择 **Project**，然后选择 Eclipse 菜单中的 **Properties** 菜单项。
2. 选择 **CheckStyle active for this project** 复选框，单击 **OK**。

图 **4.** 在 **Eclipse** 中配置 **CheckStyle** 插件



Eclipse 重新构建工作空间，并在 Eclipse 控制台中列示已发现的编码冲突，如图 5 所示：

图 **5. Eclipse** 中 **CheckStyle** 的代码冲突列表



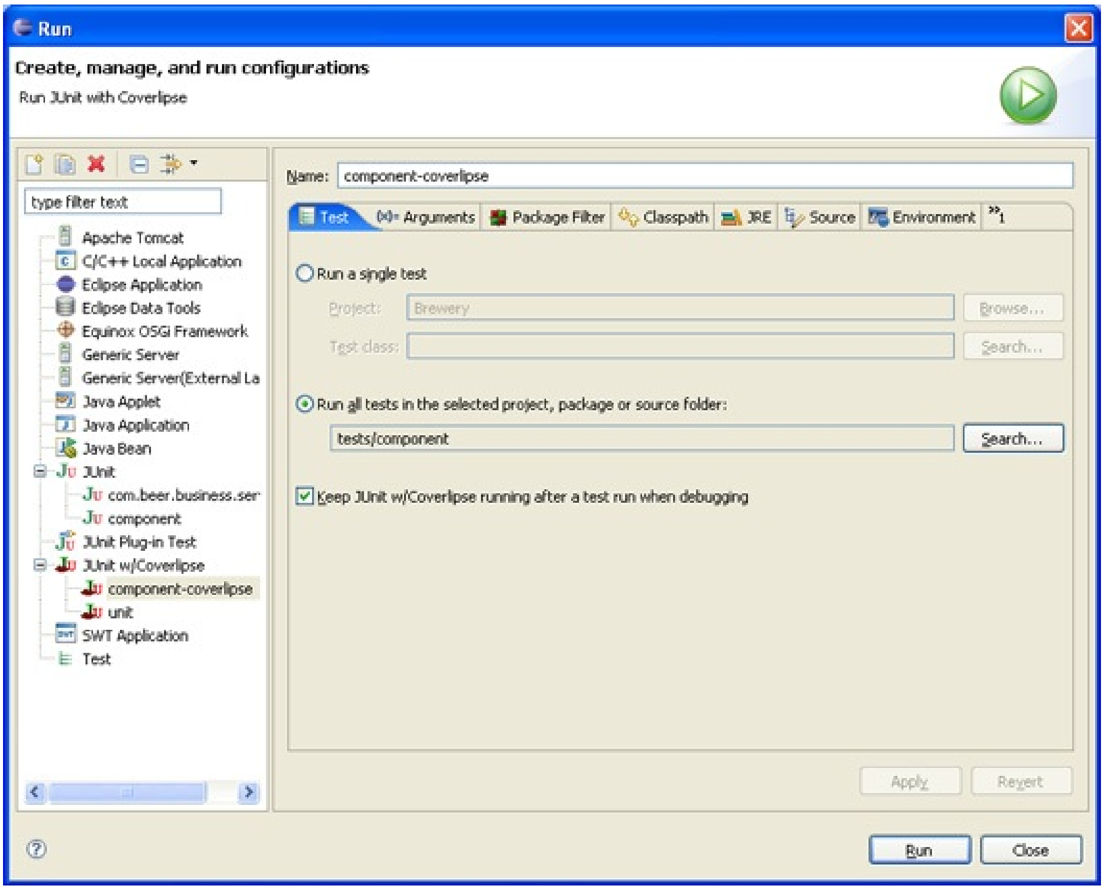
使用 CheckStyle 插件在 Eclipse 内嵌入编码标准检验是一种很棒的方法，用这种方法可以在编码时 积极地改进代码，从而在开发周期的早期发现源代码中潜在的缺陷。这么做还有更多的好处，如节省时间、减少失败，也因此会减少项目的成本。没错，这就是一种积极主动的方式！

用 Coverlipse 确认覆盖率

Coverlipse 是一个用于 Cobertura 的 Eclipse 插件，Cobertura 是一个代码覆盖率工具，可以用它来评估具有相应测试的源代码的比率。Cobertura 也提供一个 Ant 任务和 Maven 插件，但用 Cobertura，您可以在编写代码时 评估代码覆盖率。您见过这样的模式吗？

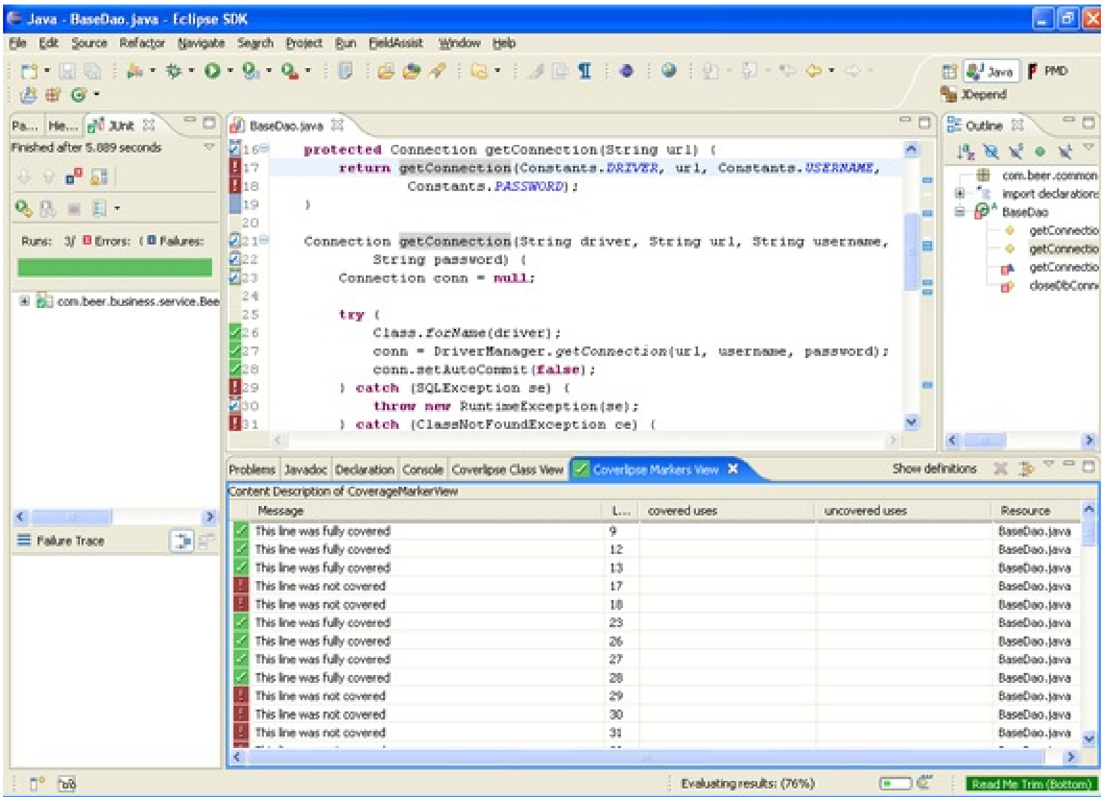
通过选择 Eclipse 菜单项 **Run** 安装 Coverlipse 插件并将其和 JUnit 关联起来，该操作会显示一系列运行配置选项，例如 JUnit、SWT 应用程序和 Java™ 应用程序。右键单击它并选择 **JUnit w/Coverlipse** 节点中的 **New**。在这里，需要确定 JUnit 测试的位置，如图 6 所示：

图 **6.** 配置 **Coverlipse** 以获取代码覆盖率



一旦单击了 **Run**，Eclipse 会运行 Coverlipse 并在源代码（如图 7 所示）中嵌入标记，该标记显示了具有相关 JUnit 测试的代码部分：

图 **7. Coverlipse** 生成的具有嵌入类标记的报告



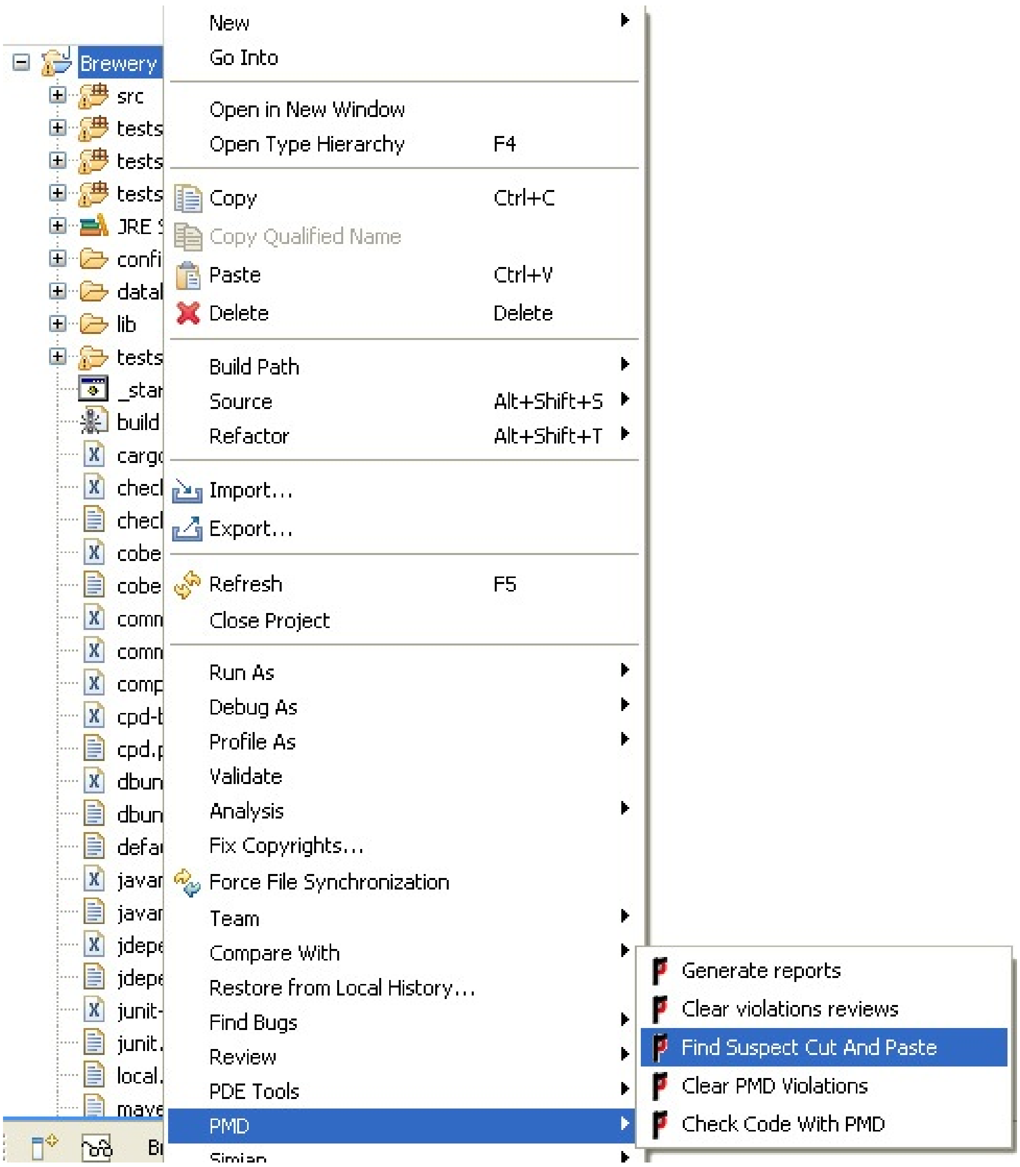
正如您所见，使用 Coverlipse Eclipse 插件可以更快地确定代码覆盖率。例如，这种实时数据功能有助于在将代码签入 CM 系统前 更好地进行测试。这对渐进编程来说意味着什么呢？

用 CPD 捕捉代码重复

Eclipse 的 PMD 插件提供了一项叫做 CPD（或复制粘贴探测器）的功能，用于寻找重复的代码。为在 Eclipse 中使用这项便利的工具，需要安装具有 PMD 的 Eclipse 插件，该插件具有 CPD 功能。

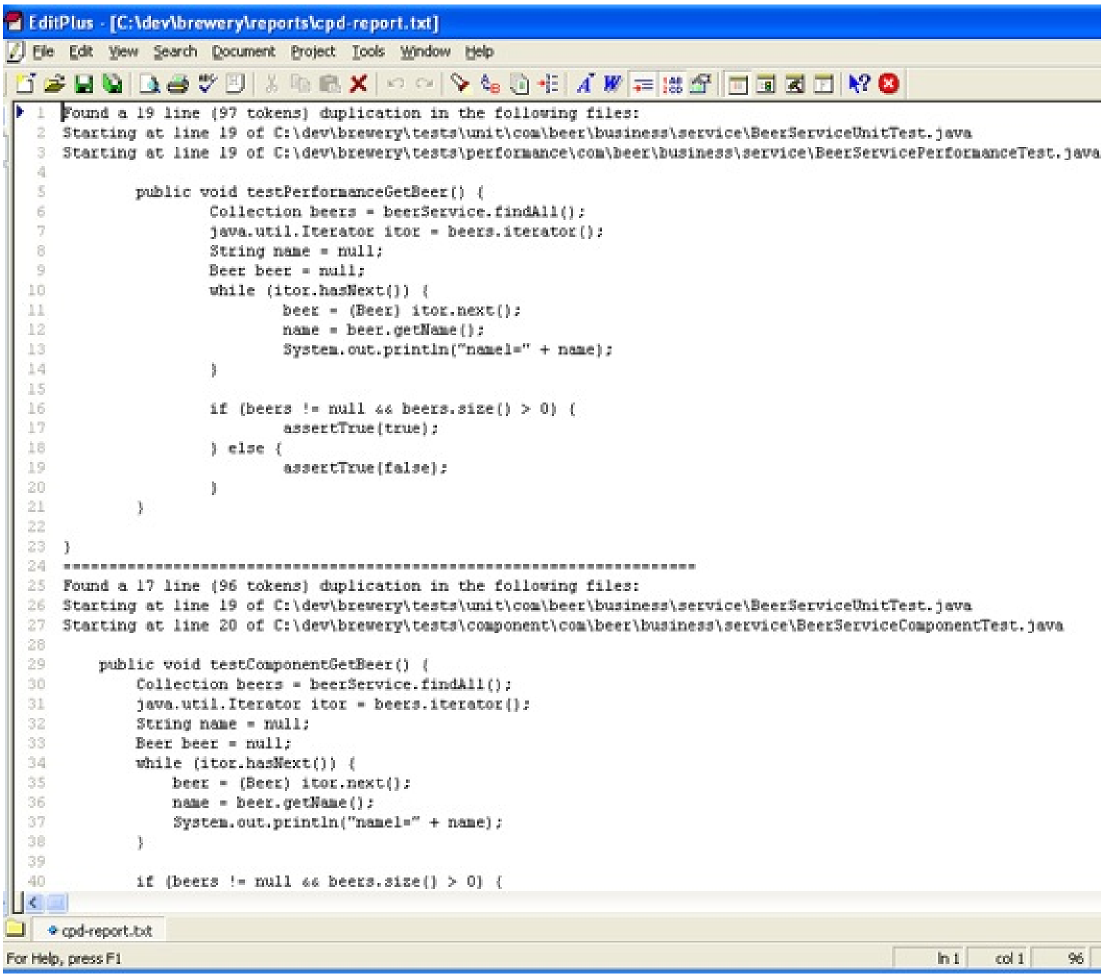
为寻找重复的代码，请用右键单击一个 Eclipse 项目并选择 **PMD | Find Suspect Cut and Paste**，如图 8 所示：

图 **8.** 使用 **CPD** 插件运行复制粘贴检验



一旦运行了 CPD，您的 Eclipse 根目录下就会创建出一个 `report` 文件夹，其中包含一个叫做 cpd.txt 的文件，文件中列示了所有重复的代码。图 9 中是一个 cpd.txt 文件的例子：

图 **9. Eclipse** 插件生成的 **CPD** 文本文件



```
Found a 19 line (97 tokens) duplication in the following files:
Starting at line 19 of C:\dev\brewery\tests\unit\com\beer\business\service\BeerServiceUnitTest.java
Starting at line 19 of C:\dev\brewery\tests\performance\com\beer\business\service\BeerServicePerformanceTest.java

        public void testPerformanceGetBeer() {
                Collection beers = beerService.findAll();
                java.util.Iterator itor = beers.iterator();
                String name = null;
                Beer beer = null;
                while (itor.hasNext()) {
                        beer = (Beer) itor.next();
                        name = beer.getName();
                        System.out.println("name1=" + name);
                }

                if (beers != null && beers.size() > 0) {
                        assertTrue(true);
                } else {
                        assertTrue(false);
                }
        }

}
=====================================================================
Found a 17 line (96 tokens) duplication in the following files:
Starting at line 19 of C:\dev\brewery\tests\unit\com\beer\business\service\BeerServiceUnitTest.java
Starting at line 20 of C:\dev\brewery\tests\component\com\beer\business\service\BeerServiceComponentTest.java

    public void testComponentGetBeer() {
        Collection beers = beerService.findAll();
        java.util.Iterator itor = beers.iterator();
        String name = null;
        Beer beer = null;
        while (itor.hasNext()) {
            beer = (Beer) itor.next();
            name = beer.getName();
            System.out.println("name1=" + name);
        }

        if (beers != null && beers.size() > 0) {
```

靠人工来寻找重复的代码是一项挑战，但使用像 CPD 这样的插件却能在编码时轻松地发现重复的代码。

使用 **JDepend** 进行依赖项检查

JDepend 是个可免费获取的开源工具，它为包依赖项提供面向对象的度量值，以此指明代码库的弹性。换句话说，JDepend 可有效测量一个架构的健壮性（反之，脆弱性）。

除了 Eclipse 插件，JDepend 还提供一个 Ant 任务、Maven 插件和一个 Java 应用程序，用以获取这些度量值。对于相同的信息，它们有着不同的传递机制；但 Eclipse 插件的特别之处和相应优点是：它能以更接近源代码（即，编码时）的方式传递这条信息。

图 10 演示了使用 Eclipse JDepend 插件的方法：通过右键单击源文件夹并选择 **Run JDepend Analysis**。一定要选择一个含源代码的源文件夹；否则看不到此菜单项。

图 **10.** 使用 **JDepend Analysis** 分析代码

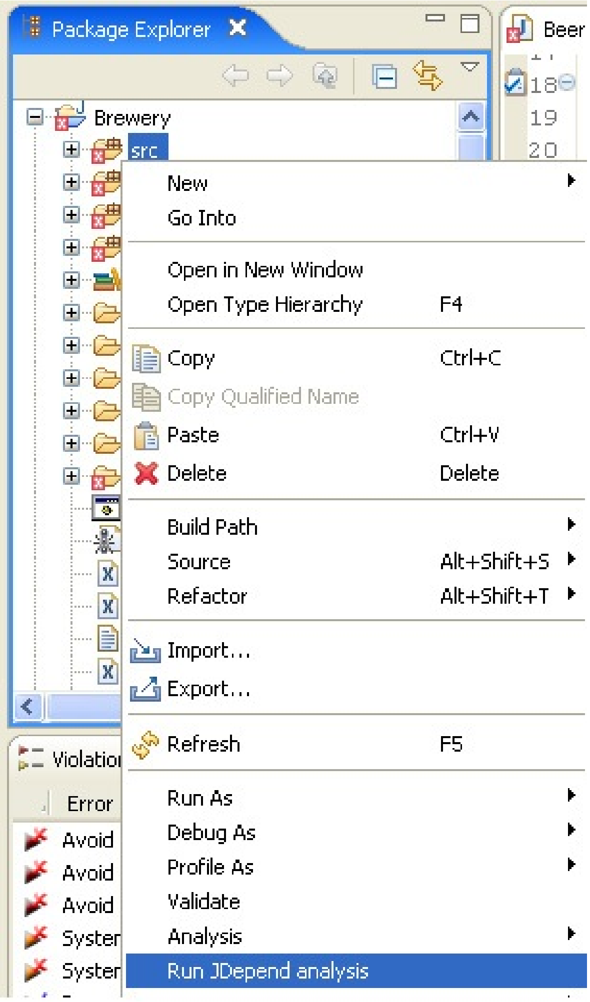
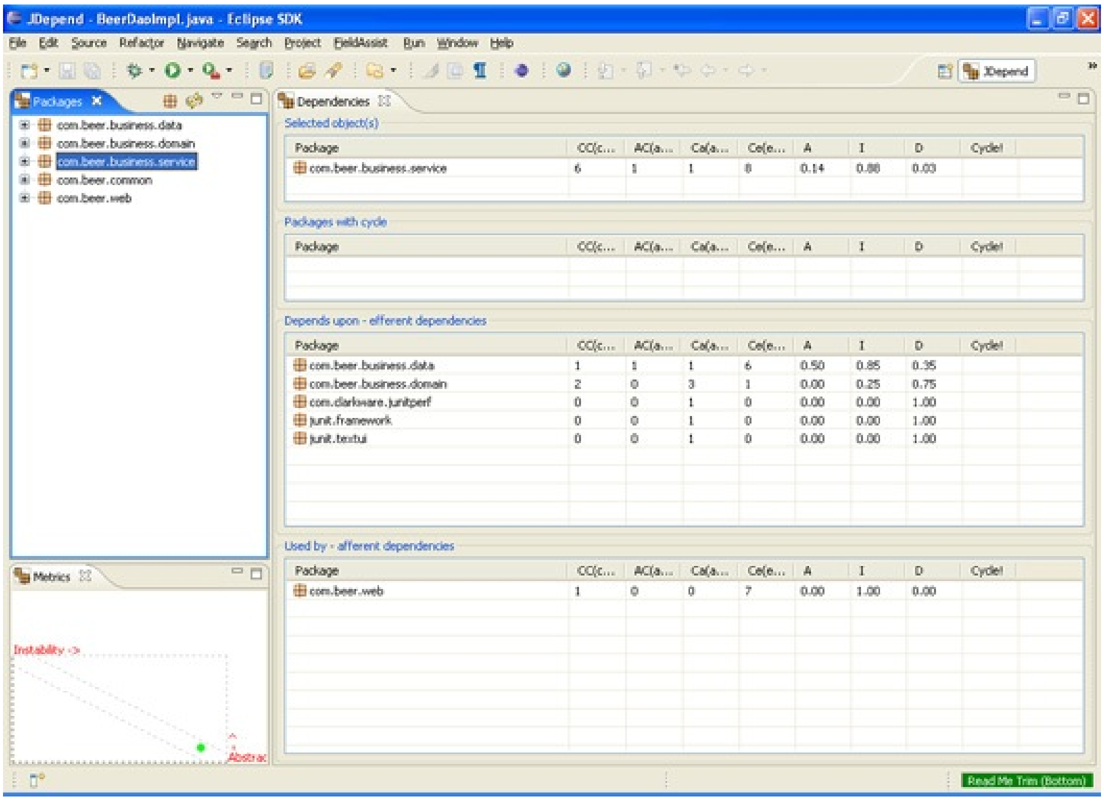


图 11 显示了运行 JDepend Analysis 时生成的报告。左边显示包，右边显示针对每个包的依赖项度量值。

图 **11. Eclipse** 项目中的包依赖项



正如您所见，JDepend 插件提供了有助于不断观察架构可维护性变化的大量信息 —— 这其中最大的好处是您可以在编码时看到这些数据。

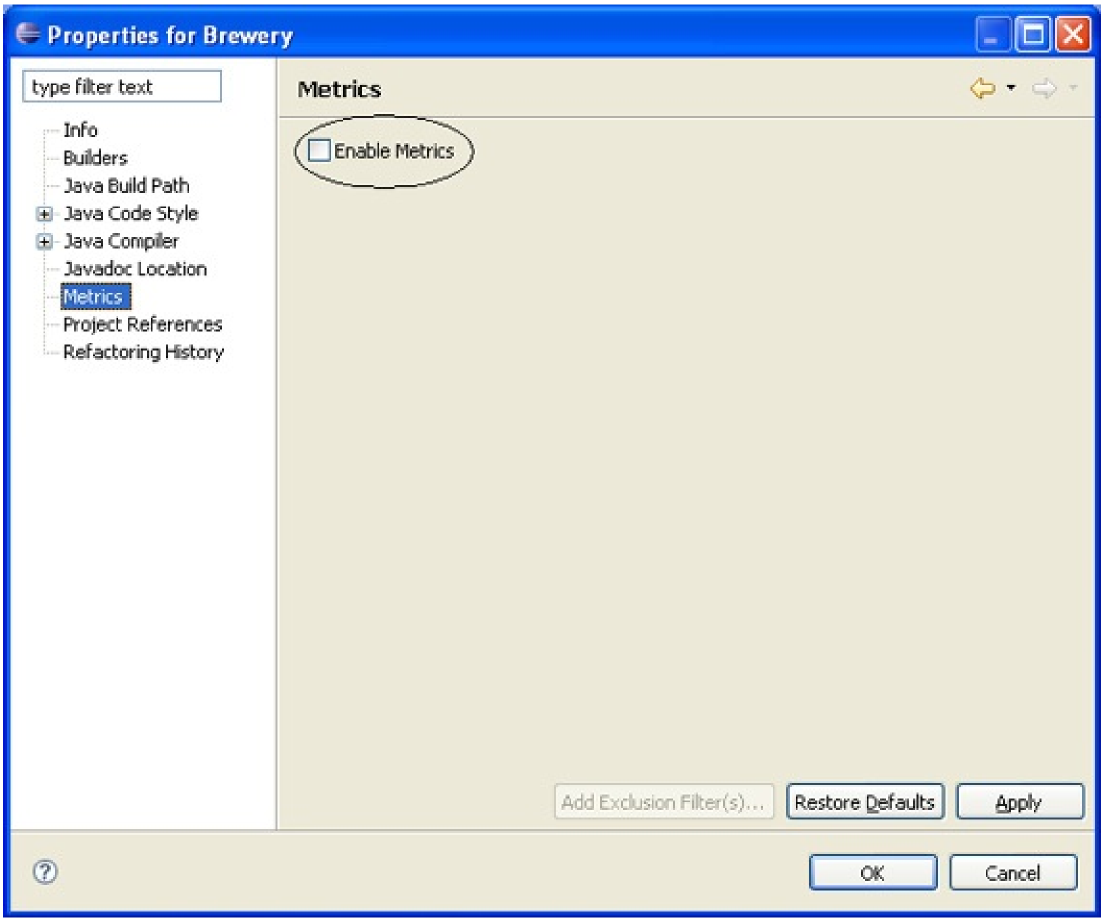
用 Metrics 测量复杂度

“五大”代码分析最后的一项是测量复杂度。Eclipse 提供一种叫做 Metrics 的插件，使用该插件可以进行许多有用的代码度量，包括圈复杂度度量，它用于测量方法中惟一路径的数目。

安装 Metrics 插件并重启 Eclipse；然后遵循下列步骤：

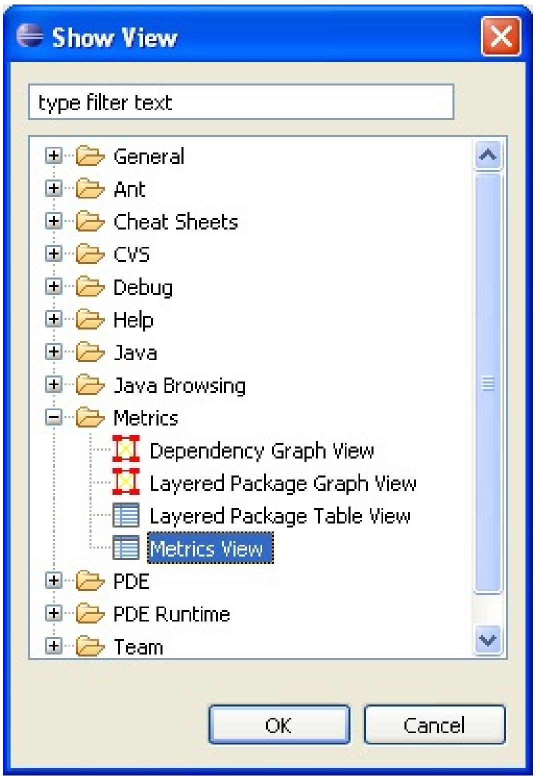
1. 右键单击您的项目并选择 **Properties** 菜单。在结果窗口中，选择 **Enable Metrics plugin** 复选框并单击 **OK**，如图 12 所示：

图 **12.** 为项目配置 **Metrics**



2. 从 Eclipse 中选择 **Window** 菜单打开 Metrics 视图，然后选择 **Show View | Other...**。
3. 选择 **Metrics | Metrics View** 打开如图 13 中显示的窗口。您需要使用 Java 透视图并重新构建项目，从而显示这些度量值。

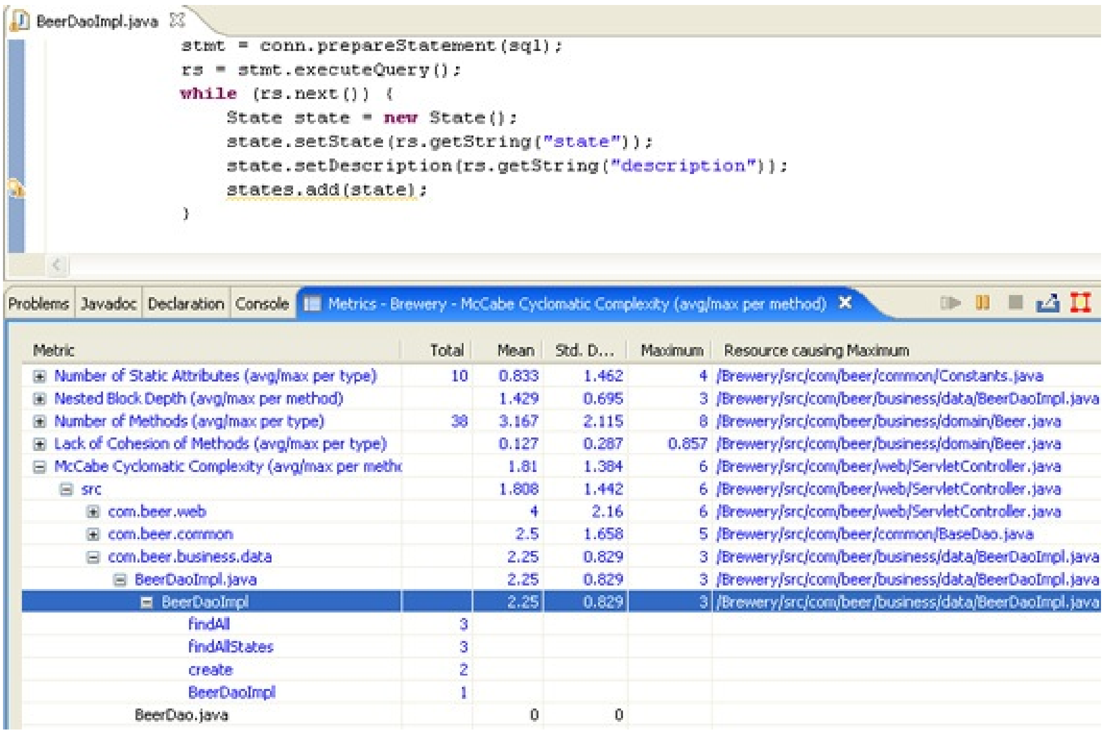
图 **13.** 打开 **Eclipse** 中的 **Metrics View**



4. 单击 **OK** 来显示如图 14 中的窗口。

在此例中，我正在查看一个单独方法的圈复杂度。真正妙的是您可以双击 Metrics 列表中的方法，该插件会在 Eclipse 编辑器中为此方法打开源代码。这就让修正变得超级简单（如果需要的话）！

图 **14.** 查看方法的圈复杂度



正如我之前提到过的，Eclipse Metrics 插件还提供了许多功能强大的度量值，有助于您在开发软件的过程中改进代码 —— 可见，它是一个渐进编程意义上的插件!

合适的才是最好的

正如您从本文中看到的那样，将“五大”测量方法，即编码标准、代码重复、代码覆盖率、依赖项分析和复杂度监控，用于改进代码质量十分重要。但适合您的才是好的。请记住还有其他许多可用的 Eclipse 插件（比如 PMD 和 FindBugs）能够帮助您在开发周期的早期改进代码质量。不管您想要的工具或偏爱的方法是什么，重要的是：行动起来去积极改进代码质量并让手工代码检验的过程变得更加有效。我估计您使用这些插件一段时间后，就再也离不开它们了。

让开发自动化: 除掉构建脚本中的气味

创建一致、可重复、可维护的构建

您把多少时间花在维护项目构建脚本上？也许远远超出您预期的或者可以忍受的时间。其实大可不必遭受如此痛苦的经历。在这一期的让开发自动化 中，开发自动化专家 Paul Duvall 将演示如何改进很多常见的妨碍团队创建一致的、可重复的、可维护的构建的实践。

当描述代码之类的东西时，我不喜欢 “气味（smell）”这个词。因为用拟人的手法来谈论比特和字节往往令人觉得很怪异。并不是说“气味”这个词不能准确地反映出某种表明代码可能有错误的症状，只是我觉得这样听起来很滑稽。然而，我依然选择再次用这种令人厌烦的方式来描述软件构建，坦白说，这是因为这些年我见过的很多构建脚本都散发着难闻的气味。

在创建构建脚本时，即使是伟大的程序员也常常会遇到困难。就好像最近才学会如何编写程序性 代码似的 —— 他们还会编写庞大的单块构建文件、通过复制-粘贴编写代码、对属性进行硬编码等等。我总是很想知道为什么会这样。也许是因为构建脚本没有被编译成客户最终会使用的东西？然而我们都知道，要创建客户最终使用的代码，构建脚本是中心，如果那些脚本败絮其中，那么要想有效地创建 软件，就需要克服重重挑战。

幸运的是，您可以轻松地在构建（不管是 Ant、Maven 还是定制的）之上部署一些实践，它们虽然可以帮助您创建一致的、可重复的、可维护的构建，但其过程会很长。学习如何创建更好的构建脚本的一种有效的方法是搞清楚哪些事情不要 去做，理解其中的道理，然后看看做事的正确 方法。在本文中，我将详细论述您应该避免的 9 种最常见的构建中的气味，为什么应该避免它们，以及如何修复它们：

- 惟 IDE 的构建
- 复制-粘贴式的编写脚本方法
- 冗长的目标
- 庞大的构建文件
- 没有清理干净
- 硬编码的值
- 测试失败还能构建成功
- 魔力机
- 格式的缺失

关于本系列

作为一名开发人员，我们的工作就是为用户将过程自动化。然而，我们当中有很多人却忽视了将我们自己的开发过程自动化的机会。为此，我编写了 让开发自动化 这个系列的文章，专门探索软件开发过程自动化的实际应用，并教您何时以及如何成功地应用自动化。

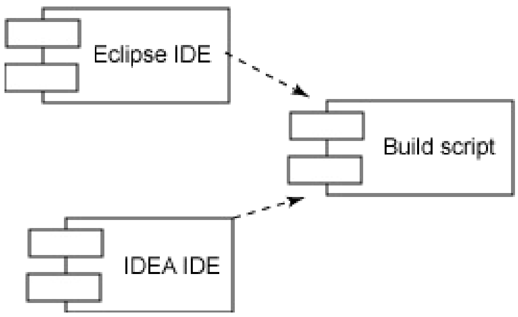
这里无意给出完整的列表，不过这份列表的确代表了近年来我读过的和写过的构建脚本中，我遇到的较为常见的一些气味。有些工具，例如 Maven，是为处理与构建有关的很多管道而设计的，它们可以帮助减轻部分气味。但是无论使用什么工具，还是有很多问题会发生。

避免惟 IDE 的构建

惟 IDE（IDE-only）的构建是指只能通过开发人员的 IDE 执行的构建，不幸的是，这似乎在构建中很常见。惟 IDE 的构建的问题是，它助长了“在我的计算机上能运行”问题，即软件在开发人员的环境中可以运行，但是在任何其他人的环境中就不能运行。而且，由于惟 IDE 构建自动化程度不是很高，因而为集成到持续集成（Continuous Integration）环境带来极大的挑战。实际上，没有人为的干预，惟 IDE 常常无法自动化。

我们要清楚：使用 IDE 来执行构建并没有错，但是 IDE 不应该成为能构建软件的惟一环境。特别是，一个完全用脚本编写的构建，可以使开发团队能够使用多种 IDE，因为只存在从 IDE 到构建的依赖性，而不存在相反方向的依赖性，如图 1 所示：

图 **1. IDE** 与构建的依赖关系



惟 IDE 的构建有碍自动化，清除的惟一方法就是创建可编写脚本的构建。有足够的文档和太多的书籍可以为您提供指导（见 参考资料），而像 Maven 之类的项目也为从头开始定义构建提供了极大的方便。不管采用何种方法，都是选择一种构建平台，然后尽快地让项目成为可编写脚本的。

复制-粘贴就像廉价的香水

复制代码是软件项目当中一个常见的问题。实际上，甚至很多流行的开放源码项目都存在 20% 到 30% 的复制代码。代码复制令软件程序更难于维护，同理，构建脚本中的复制代码也存在这样的问题。例如，想象一下，假设您需要通过 Ant 的 `fileset` 类型引用特定的文件，如清单 1 所示：

清单 **1.** 复制-粘贴 **Ant** 脚本

```
<fileset dir="./brewery/src" >
  <include name="**/*.java"/>
  <exclude name="**/*.groovy"/>
</fileset>
```

如果需要在其他地方引用这组文件，例如为了编译、检查或生成文档，那么最终您可能会在多个地方使用相同的 `fileset`。如果在将来某个时候，您需要对那个 `fileset` 做出修改（比如说排除 `.groovy` 文件），那么最终可能需要在多个地方做更改。显然，这不是可维护的解决方案。然而，要除掉这股气味其实很简单。

如清单 2 所示，通过 Ant 的 `patternset` 类型可以引用一个逻辑名称，以表示所需要的文件。那么，当需要向 `fileset` 添加（或排除）文件时，只需更改一次。

清单 **2.** 复制**-**粘贴 **Ant** 脚本

```
<patternset id="sources.pattern">
  <include name="**/*.java"/>
  <exclude name="**/*.groovy"/>
</patternset>
...
<fileset dir="./brewery/src">
  <patternset refid="sources.pattern"/>
</fileset>
```

对于精通面向对象编程的人来说，这种修复方法看上去很熟悉：既定的惯例不是在不同的类中一次又一次地定义相同的逻辑，而是将那个逻辑放在一个方法中，在不同地方都可以调用这个方法。于是，这个方法成为惟一的维护点，从而可以限制错误级联并可以鼓励重用。

不要掺入冗长目标的气味

Martin Fowler 在他撰写的 *Refactoring* 这本书中，对代码中存在冗长方法的气味这个问题做了精妙的描述 —— 过程越长，越难理解。实际上，冗长方法最终会担负太多的责任。当谈到构建时， 冗长目标这种构建气味是指更难于理解和维护的脚本。清单 3 就展示了一个相当冗长的目标：

清单 **3.** 冗长目标

```
<target name="run-tests">
  <mkdir dir="${classes.dir}"/>
  <javac destdir="${classes.dir}" debug="true">
    <src path="${src.dir}" />
    <classpath refid="project.class.path"/>
  </javac>
  <javac destdir="${classes.dir}" debug="true">
    <src path="${test.unit.dir}"/>
    <classpath refid="test.class.path"/>
  </javac>
  <mkdir dir="${logs.junit.dir}" />
  <junit fork="yes" haltonfailure="true" dir="${basedir}" printsummary="yes">
    <classpath refid="test.class.path" />
    <classpath refid="project.class.path"/>
    <formatter type="plain" usefile="true" />
    <formatter type="xml" usefile="true" />
    <batchtest fork="yes" todir="${logs.junit.dir}">
      <fileset dir="${test.unit.dir}">
        <patternset refid="test.sources.pattern"/>
      </fileset>
    </batchtest>
  </junit>
  <mkdir dir="${reports.junit.dir}" />
  <junitreport todir="${reports.junit.dir}">
    <fileset dir="${logs.junit.dir}">
      <include name="TEST-*.xml" />
      <include name="TEST-*.txt" />
    </fileset>
    <report format="frames" todir="${reports.junit.dir}" />
  </junitreport>
</target>
```

这个冗长的目标（相信我，我还见过冗长得多的目标）要执行四个不同的过程：编译源代码、编译测试、运行 JUnit 测试和创建一个 JUnitReport。要担负的责任已经够多了，更不用说将所有 XML 放在一个地方所增加的相关的复杂性。实际上，这个目标可以拆分成四个不同的、逻辑上的目标，如清单 4 所示：

清单 **4.** 提取目标

```
<target name="compile-src">
   <mkdir dir="${classes.dir}"/>
   <javac destdir="${classes.dir}" debug="true">
     <src path="${src.dir}" />
     <classpath refid="project.class.path"/>
   </javac>
 </target>

 <target name="compile-tests">
   <mkdir dir="${classes.dir}"/>
   <javac destdir="${classes.dir}" debug="true">
     <src path="${test.unit.dir}"/>
     <classpath refid="test.class.path"/>
   </javac>
 </target>

 <target name="run-tests" depends="compile-src,compile-tests">
   <mkdir dir="${logs.junit.dir}" />
   <junit fork="yes" haltonfailure="true" dir="${basedir}" printsummary="yes">
     <classpath refid="test.class.path" />
     <classpath refid="project.class.path"/>
     <formatter type="plain" usefile="true" />
     <formatter type="xml" usefile="true" />
     <batchtest fork="yes" todir="${logs.junit.dir}">
       <fileset dir="${test.unit.dir}">
         <patternset refid="test.sources.pattern"/>
       </fileset>
     </batchtest>
   </junit>
 </target>

 <target name="run-test-report" depends="compile-src,compile-tests,run-tests">
     <mkdir dir="${reports.junit.dir}" />
     <junitreport todir="${reports.junit.dir}">
     <fileset dir="${logs.junit.dir}">
       <include name="TEST-*.xml" />
       <include name="TEST-*.txt" />
     </fileset>
     <report format="frames" todir="${reports.junit.dir}" />
   </junitreport>
 </target>
```

可以看到，由于每个目标只担负一种责任，清单 4 中的代码理解起来要容易得多。根据用途分离目标，不但可以减少复杂性，还为在不同上下文中使用目标创造了条件，必要时还可以重用。

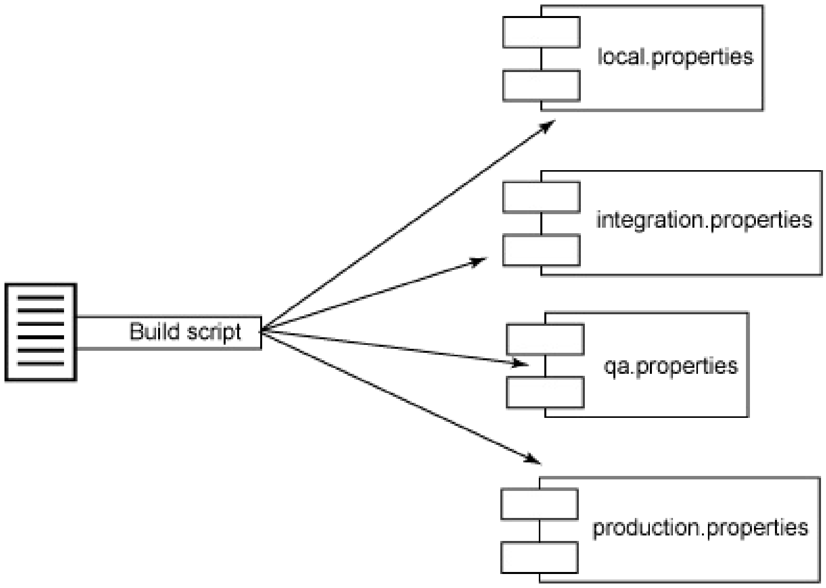
庞大的构建文件也有一种很重的气味

Fowler 还将 庞大的类也看作一种代码气味。就构建脚本而言，有这种类似气味的就是庞大的构建文件，它相当难以读懂。很难知道哪个目标是做什么的，目标的依赖关系是什么。这同样会给维护带来问题。而且，庞大的构建文件通常有相当多的剪切-粘贴的痕迹。

为了缩小构建文件，可以从脚本中找出逻辑上相关的部分，将它们提取到更小的构建文件中，由主构建文件来执行这些较小的构建文件（例如，在 Ant 中，可以使用 `ant` 任务调用其他构建文件）。

通常，我喜欢根据核心功能拆分构建脚本，确保它们可以作为独立脚本来执行（想想构建组件化）。例如，在我的 Ant 构建中，我喜欢定义四种类型的开发者测试：单元、组件、系统和功能。而且，我还喜欢运行四种类型的自动检查工具：编码标准、依赖性分析、代码覆盖范围和代码复杂度。我不是将这些测试和检查工具的执行放在一个庞大的构建脚本中（还加上编译、数据库集成和部署），而是将测试和检查工具的执行目标提取到两个不同的构建文件中，如图 2 所示：

图 **2.** 提取构建文件



更小、更简洁的构建文件维护和理解起来要容易得多。实际上，这种模式对于代码而言同样适用。我们似乎在这里看到了模式的概念，不是吗？

没有清理

没有严格减少所有底层假设的构建无疑是一颗定时炸弹。例如，如果构建没有避免一些简单的假设，例如会去掉用陈旧的数据生成的二进制文件，那么前一次构建遗留下来的文件就会引起错误。或者，正是由于前一次构建留下的文件，构建竟然得以"成功"，这种情况更糟糕。

幸运的是，这个问题的解决办法很直观：只需删除任何之前的构建留下的所有目录和文件，就可以很容易地消除假设。这个简单的动作就可以减少假设，保证构建的成功或失败都是正确的。清单 5 演示了通过使用 `delete` Ant 任务删除之前的构建所使用的所有文件或目录，从而清理构建环境的一个例子：

清单 **5.** 事先清理

```
<target name="clean">
  <delete dir="${logs.dir}" quiet="true" failonerror="false"/>
  <delete dir="${build.dir}" quiet="true" failonerror="false"/>
  <delete dir="${reports.dir}" quiet="true" failonerror="false"/>
  <delete file="cobertura.ser" quiet="true" failonerror="false"/>
</target>
```

众所周知，旧的构建遗留下来的文件会导致很多不必要的麻烦。为了自己的方便，在运行一个构建之前，务必先删除构建所创建的任何工件。

硬编码的臭味

复制-粘贴式的编程有碍重用，将值进行硬编码又何尝不是呢。当构建脚本包含硬编码的值时，如果某个方面需要修改，那么就需要在多个地方修改那个值。更糟糕的是，很可能会忽略了某个地方而没有改那个值，从而引起与不匹配的值相关的错误，这种错误是很隐蔽的。而且，如果相信我的建议，选择使用多个构建脚本，那么硬编码的值将可能会成为构建维护中最终的挑战。在这一点上也请相信我！

例如，在清单 6 中， `run-simian` 任务有很多硬编码的路径和值，即 `_reports` 目录：

清单 **6.** 硬编码的值

```
  <target name="run-simian">
    <taskdef resource="simiantask.properties"
      classpath="simian.classpath" classpathref="simian.classpath" />
    <delete dir="./_reports" quiet="true" />
    <mkdir dir="./_reports" />
    <simian threshold="2" language="java"
      ignoreCurlyBraces="true" ignoreIdentifierCase="true" ignoreStrings="true"
      ignoreStringCase="true" ignoreNumbers="true"  ignoreCharacters="true">
      <fileset dir="${src.dir}"/>
      <formatter type="xml" toFile="./_reports/simian-log.xml" />
    </simian>
    <xslt taskname="simian"
      in="./_reports/simian-log.xml"
      out="./_reports/Simian-Report.html"
      style="./_config/simian.xsl" />
  </target>
```

如果硬编码 `_reports` 目录，那么当我决定将 Simian 报告放到另一个目录时，就会很麻烦。而且，如果其他工具在脚本的其他地方使用这个目录，那么很可能会有人输错目录名称，导致报告显示在不同的目录中。这时可以定义一个属性值，由这个属性值指向这个目录。然后，在整个脚本中都可以引用这个属性，这意味着当需要更改的时候，只需光顾一个地方，即属性的定义。清单 7 展示了重构之后的 `run-simian` 任务：

清单 **7.** 使用属性

```
  <target name="run-simian">
    <taskdef resource="simiantask.properties"
      classpath="simian.classpath" classpathref="simian.classpath" />
    <delete dir="${reports.simian.dir}" quiet="true" />
    <mkdir dir="${reports.simian.dir}" />
    <simian threshold="${simian.threshold}" language="${language.type}"
      ignoreCurlyBraces="true" ignoreIdentifierCase="true" ignoreStrings="true"
      ignoreStringCase="true" ignoreNumbers="true"  ignoreCharacters="true">
      <fileset dir="${src.dir}"/>
      <formatter type="xml" toFile="${reports.simian.dir}/${simian.log.file}" />
    </simian>
    <xslt taskname="simian"
      in="${reports.simian.dir}/${simian.log.file}"
      out="${reports.simian.dir}/${simian.report.file}"
      style="${config.dir}/${simian.xsl.file}" />
  </target>
```

硬编码的值不仅没有提高灵活性，反而拟制了灵活性。就像在源代码中很容易硬编码数据库连接 `String` 一样，在构建脚本中也应该避免将路径之类的东西硬编码。

测试失败时，构建却能成功

构建远远不止于单纯的源代码编译，它还可能包括自动化开发者测试的执行，如果想让软件一直正常运行，那么决不能允许构建中有任何失败的测试。别忘了，如果测试都得不到信任，那么还要测试干什么呢？

清单 8 是这种构建气味的一个例子。注意 `junit` Ant 任务的 `haltonfailure` 属性被设置为 `false`（它的缺省值）。这意味着即使任何 JUnit 测试是失败的，构建也不会失败。

清单 **8.** 气味：测试失败，构建却成功

```
<junit fork="yes" haltonfailure="false" dir="${basedir}" printsummary="yes">
  <classpath refid="test.class.path" />
  <classpath refid="project.class.path"/>
  <formatter type="plain" usefile="true" />
  <formatter type="xml" usefile="true" />
  <batchtest fork="yes" todir="${logs.junit.dir}">
  <fileset dir="${test.unit.dir}">
    <patternset refid="test.sources.pattern"/>
  </fileset>
  </batchtest>
</junit>
```

有两种方法防止构建中的这种气味。第一种方法是将 `haltonfailure` 属性设置为 `true` 。这样就可以防止测试失败构建却成功的情况发生。

对于这种方法，我惟一不喜欢的地方是，我想看看有多大百分比的测试遭到了失败，以便弄清楚失败的模式。因此第二种方法就是，每当有测试失败，就设置一个属性。然后，我对 Ant 进行配置，使得当执行了所有的测试之后，构建最终失败。这两种方法都行之有效。清单 9 演示了使用 `tests.failed` 属性的第二种方法：

清单 **9.** 测试令构建失败

```
<junit dir="${basedir}" haltonfailure="false" printsummary="yes"
  errorProperty="tests.failed" failureproperty="tests.failed">
  <classpath>
    <pathelement location="${classes.dir}" />
  </classpath>
  <batchtest fork="yes" todir="${logs.junit.dir}" unless="testcase">
    <fileset dir="${src.dir}">
      <include name="**/*Test*.java" />
   </fileset>
  </batchtest>
  <formatter type="plain" usefile="true" />
  <formatter type="xml" usefile="true" />
</junit>
<fail if="tests.failed" message="Test(s) failed." />
```

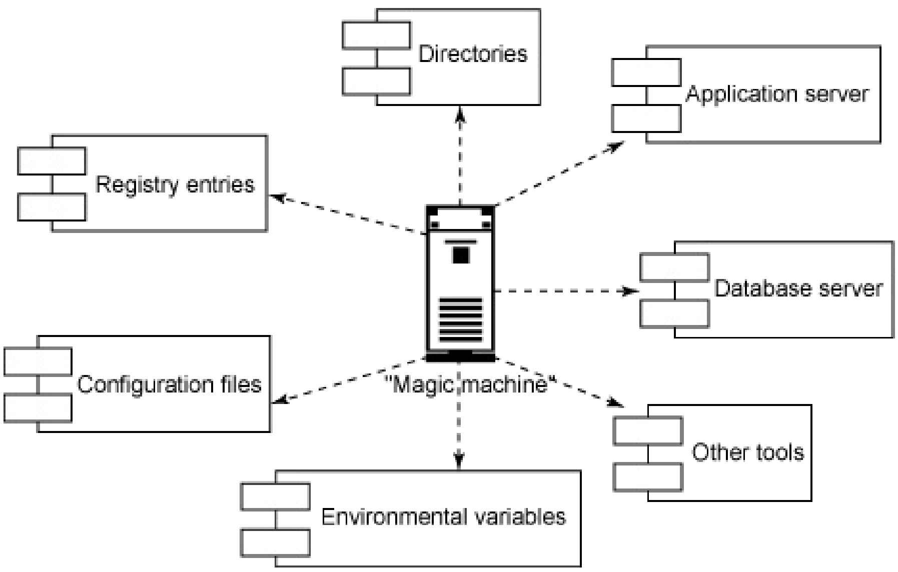
如果测试失败时构建还能通过，就会提供关于安全性的一种错感。如果测试失败，那么让构建也失败：早一点从容地处理问题，总比以后问题半夜三更把您从梦中唤醒要好。

魔力机的气味

在本文谈到的所有气味当中，这一种也许是最难闻的，因为魔力机（*magic machine*）是那种刚好惟一能够构建一个公司的软件应用程序的硬件。这种情况看上去难以相信，实则不然。在我的职业生涯中，就多次碰到过它。当依赖性丢失，或者当不断累积的问题爆发时，这些机器就获得了所谓的魔力。

我们很容易看出，公司基础设施中的一台正常的机器是如何获得魔力的：随着时间的推移，开发者无意间在机器的脚本中添加了硬性的依赖性，包含了对目录路径的全限定引用，甚至安装了只有一台机器上有的工具，久而久之，构建在任何其他机器上再也不能运行了。图 3 就展示了一个例子：

图 **3.** 魔力机



对一台机器的硬编码引用，包括特定驱动器（例如 C：）的路径，以及机器上特有的工具，都是令一台机器着魔的罪魁祸首。每当看到对 C: 盘的引用，或者看到对特定工具（例如 `grep`）的调用时，应该马上更改脚本。如果发现自己声称 "`C:\Program Files\` 目录在每台机器上都有" 的时候，也要三思。

不良格式也有气味

和主流语言中的编程格式一样，在管理构建脚本的时候，也有类似的考虑。当为构建脚本考虑编程格式的时候，需要考虑以下几个方面：

- 属性名称
- 目标名称
- 目录名称
- 环境变量名称
- 缩进
- 代码行长度

就个人而言，对于格式上的约定，我喜欢尽可能利用他人的规则。幸运的是，有人已经提供了那样的参考，即 *The Elements of Ant Style*（见 参考资料）。在这本书中，作者描述了各种规则，例如用小写字母加上用于分隔单词的连字符来命名目标，以及代码行长度和缩进等。不管选择哪一种方法，始终如一地应用有关格式的规则有助于构建文件的长期维护。

构建从来没有如此好闻

我尚能忍受廉价香水的气味。但是，如果说有一样东西我无法忍受的话，那一定是难于维护的构建脚本所散发出的气味。差劲的代码显然会浪费您宝贵的时间，设计不良的构建也不例外。如果构建中还飘散着不一致的、不可重复的和不可维护的气味，那么现在就花时间重构这些至关重要的资源吧。您的开发环境定会香如玫瑰。

让开发自动化: 选择持续集成服务器

对开源 *CI* 服务器：*CruiseControl*、*Luntbuild* 和 *Continuum* 的调查

由于有许多持续集成服务（CI）服务器可以选择，所以很难决定哪个适应自己。在 让开发自动化 系列的第二篇文章中，开发自动化专家 Duvall 采用一致的评估标准和很多说明性示例，介绍了一些开源 CI 服务器，包括 Continuum、CruiseControl 和 Luntbuild。

在我脑海里，我至少能想到 12 种在当前市场上可用的 CI 服务器，包括商业的和开源的。虽然它们都试图自动进行软件构建的过程，但是都有各自的优点和不足。而且，有太多工具可供选择的不良后果就是很难决定究竟应该选择使用哪个。

在选用自动化过程的工具时，要时刻记住的就是：工具要 确实适用。选择错误的工具可能会限制整体的灵活性，会导致执行简单动作反而需要更长时间，或者会把人锁定在特定的支持工具或过程。

选择 CI 服务器的标准

通常，对一个新工具的决策分析可以归结如下：

> 我听说 Tim 在使用 Acme Inc 的工具，而且我认为 Tim 是个聪明人。所以，我也要使用 Acme Inc 的工具。现在我也是个聪明人了。

关于本系列

作为开发人员，我们的工作就是为用户提供自动化处理；但是，我们中的许多人却忽视了自动化自己的开发过程的机会。出于这个目的，让开发自动化 这一系列的文章专门研究了自动化软件开发过程的实际应用，并教您 什么时候 和 如何 成功应用自动化。

反过来，如果您问 Tim 为什么 他选择使用 Acme Inc 的工具，您可能会发现是他的公司强制要求使用的。这就是为什么重要的是要根据 自己的 具体技术和政策需求对工具进行分析。如果不这么做，可能就会选择到不符合需求的工具，甚至更糟糕的是，不能带来任何帮助的工具。

在决策的时候，通常多数人都会把重点放在工具的特性上。但是要记住，虽然特性的确重要，但还有其他指标需要考虑。在我的实践中，我发现以下五个指标在评估工具时最有帮助：

什么是持续集成？

持续集成（CI）是一种实践，可以让团队在持续的基础 上收到反馈并进行改进，不必等到开发周期后期才寻找和修复缺陷。诸如 CruiseControl 之类的检查工具是在后台运行的，它们轮询版本控制存储库，从中寻找更改之处。当发现某一更改时，这类工具就会通过 Ant 执行预定义的构建脚本。持续检查借助持续集成的实践得以改进。

- 特性
- 可靠性
- 寿命
- 目标环境
- 易用性

而且不要忘记，客观地 检查这五个方面也是重要的。

产品特性

说到 CI 服务器的特性，应当考虑该工具与版本控制系统的集成、处理构建平台（例如 Ant 和 Maven）的能力以及提供反馈和报告的能力。而且不要忘记检查其他特性，例如构建标号和管理项目的依赖项。最后，在需要做一些特定的增强时，理解产品的可扩展性会很有帮助。

表 1 详细说明了每个特性：

表 **1.** 详细的 **CI** 服务器评估特性

特性	解释
版本控制系统集成	如果工具不支持您所使用的特定版本控制系统，您真的会为它编写一个定制集成么？
构建工具集成	在选择 CI 服务器时，需要考虑目前或将要使用哪个构建工具。对于 Java™ 编程，有两个自然的选择：Ant 和 Maven，几乎所有 CI 工具都支持它们。如果构建系统既不是 Ant 也不是 Maven，那么 CI 工具支持从命令行运行程序的功能么？
反馈和报告	想想老话"如果树倒在森林中，能有人听到么？"如果构建失败，会有人知道么？如果没人知道，那么使用 CI 工具的目的是什么？所有的 CI 工具都提供一些通知机制，但是哪个最适合您呢？电子邮件？即时消息？RSS？
标号	有些开发团队喜欢跟踪构建，给构建一个唯一的标号，这样日后就能找到具体的构建实例。如果这对您来说很重要，那么要注意只有少数 CI 服务器提供了这个功能。
项目依赖项	某些情况下，在构建了一个项目之后，可能需要构建其他依赖项目。有些 CI 服务器支持这个特性，有些不支持。
易于扩展	扩展工具当前的功能有多容易？是否用插件就可以实现简单的扩展，还是总得修改代码？

从特性的角度来说，以上提到的几点在选择所需要的正确的 CI 服务器时，至关重要。

产品可靠性

因为下载和使用开源 CI 服务器很简单，所以可以试用产品来判断它的可靠性。而且，在工具的可靠性和它在市场上的时间之间，通常存在一些相关性。使用新产品时，就会冒着有未发现的 bug 的风险。而且，用户群是发现工具出现的问题的优秀资源。大量的问题贴子或者过多的复杂问题，就表示用户对这个工具的意见较大。

因为我这里讨论的服务器是开源的，所以很容易发现下载的人数，这也会是产品健康程度的一个指示。用户少可能意味着反馈渠道少，可能需要换个地方看看。

寿命前景

在下载 CI 服务器之前，了解这个服务器未来的前景会有帮助。简单地说，使用已经死亡或正走向死亡的产品不是个好主意。可以检查该工具已经出现了多少年、在它的用户群中是否有正常数量的活动。就像可以从用户群来判断产品的可靠性一样，活跃的社区是工具未来前景良好的征兆。

目标环境

CI 服务器不能在 所有 环境下工作。需要考虑服务器支持哪个操作系统以及具体的系统需求。例如，如果工具是用最新版本的 Python 编写的，那么需要确定这个版本 Python 能够用于自己的操作系统。

易用性

产品的易用性可能是最主观的指标。有些人愿意手工修改配置文件，而有些人想让所有管理任务都在应用程序中执行，例如 Web 控制台。有些服务器要求从一个屏幕单击到下一个屏幕来执行简单的管理，而其他服务器则提供了直观的向导。

如果想理解 CI 服务器的具体细节，那么漂亮的管理 Web 表单就不重要了；但是，如果人手不足、工作繁忙，那么可能不会想在管理 CI 服务器上花太多时间。

记住我在这节讨论的五个方面，再来看一下三个 CI 服务器：Apache 的 Continuum、CruiseControl 和构建管理服务器 Luntbuild。

Apache Continuum

Continuum 是最新的 CI 服务器之一，也是值得关注的一个新进入者。Continuum 的安装和配置很简单：只要下载和释放 ZIP 文件，运行命令行程序，就可以运行了。基于 Web 的界面使得配置项目很容易。而且，还不需要安装 Web 服务器，因为 Continuum 内置了 Jetty Web 服务器。并且，Continuum 可以作为 Windows 服务运行，还在应用程序的某些部分嵌入了上下文敏感的文档，从而提供了很多帮助。

想要更多细节信息？

面对如此之多 CI 服务器可以选择，本文可以引导您更详细地研究每个服务器，并决定哪个最合适。因为我比较了三个不同的服务器，所以我没有深入每个服务器的特定细节。我只是把重点放在了这些服务器安装后就提供的选项上。如果需要更多信息，请参考每个服务器的安装和配置指南。

易于使用

在使用 Continuum 时会注意到的第一件事就是它的易用性。能够在几分钟之内就把服务器运行起来并让它去查询修改。实际上，在 Windows 上启用 Continuum 只需要四步：

1. 下载 Continuum ZIP 文件（请参阅 参考资料）。
2. 把文件的内容释放到本地目录。
3. 运行 run.bat 文件，然后运行 InstallService.bat。
4. 打开浏览器指向 http://localhost:8080/。

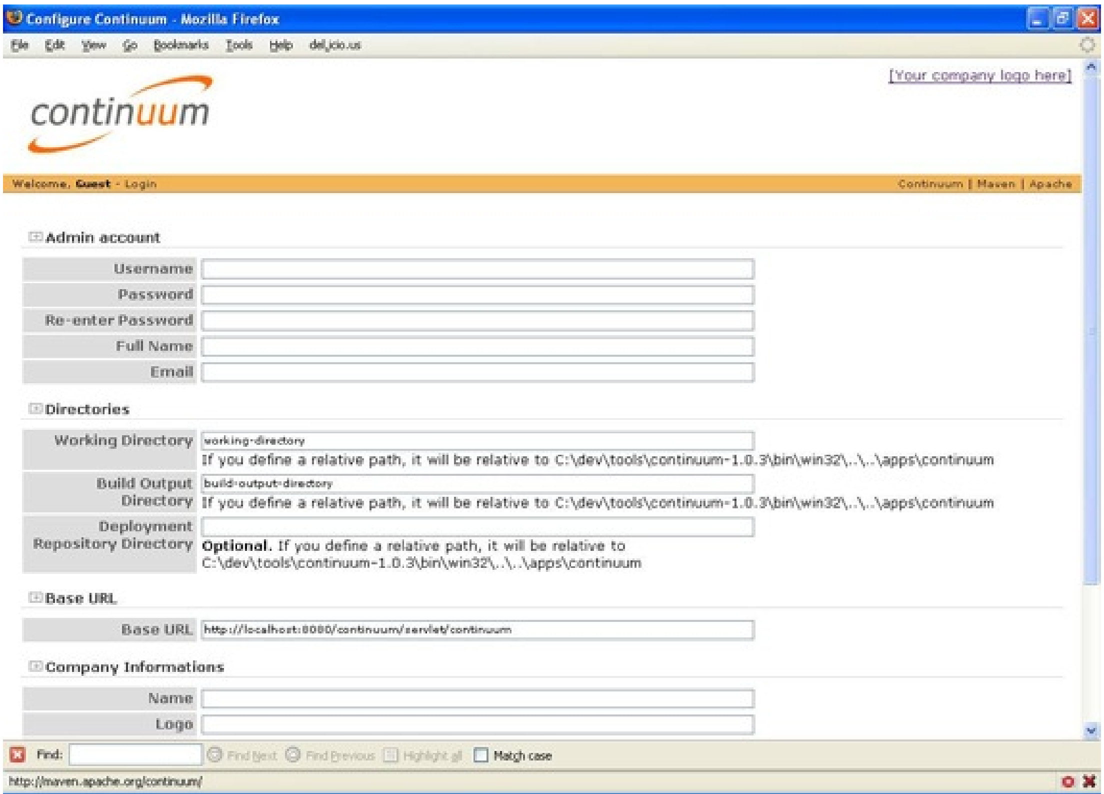
Continuum 内置支持五个版本控制系统：Subversion、CVS、StarTeam、Bazaar 和 Perforce。也部分地支持其他版本控制工具，例如 Visual Source Safe 和 ClearCase。Continuum 还支持四种构建机制：Ant、Maven1、Maven2 和 Shell（命令行）。

配置 Continuum

在第一次访问 Continuum Web 应用程序时，默认是 *guest* 帐户。guest 提供了对所有项目的只读存取，没有管理或配置项目的能力。但是，可以很容易地创建 Administrative 用户，然后设置一些适用于所有项目的属性。

图 1 显示了管理页面，它提供了管理所有项目的 Continuum 设置的能力，包括创建 Admin 帐户、构建的输出和部署目录：

图 **1. Continuum** 的配置很简单

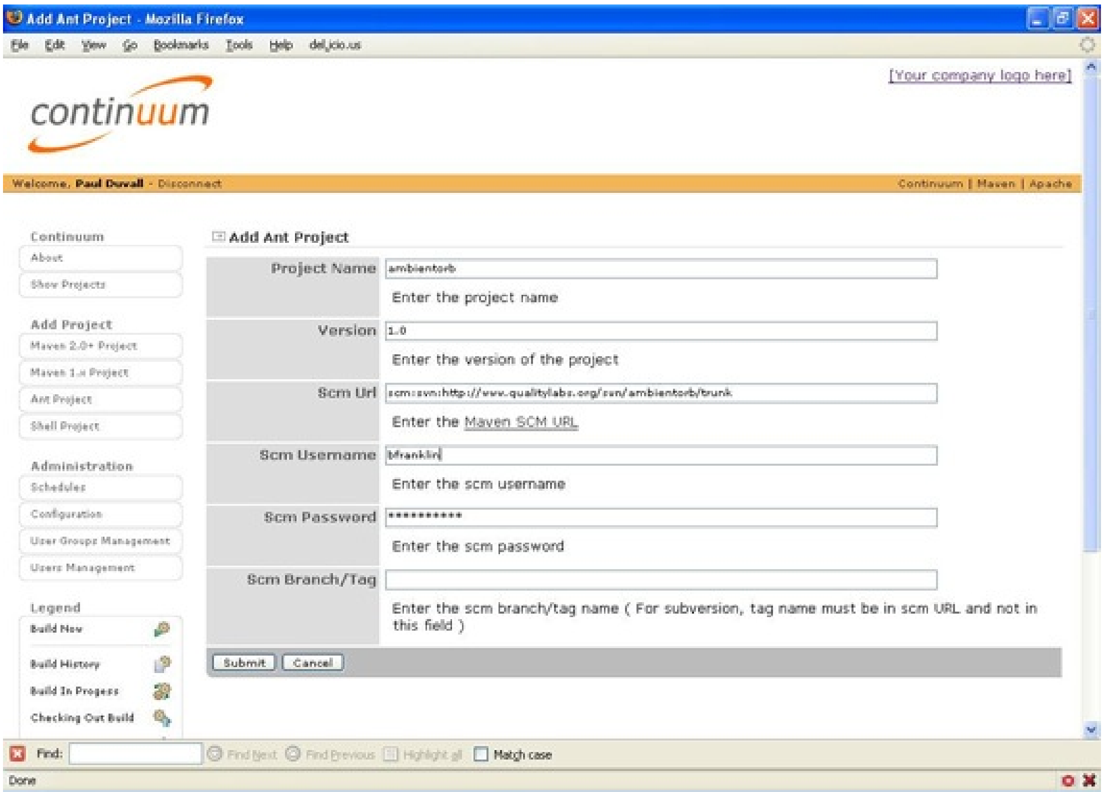


把项目添加到监视器

对 Continuum 进行配置让它监视项目也非常简单。简单到仅仅是选择期望的构建平台，例如 Ant 或 Maven2，然后把 Continuum 指到期望的版本控制系统。

图 2 显示了设置 Ant 项目时需要填充的字段：

图 **2.** 在 **Continuum** 中创建项目



在保存了这个信息之后，Continuum 每小时查询版本控制系统一次。可以修改项目的设置，查询得更频繁或更少些。我们在这里谈到的是 持续 集成，我建议每五 分钟检查修改一次，而不要每小时一次。

默认情况下，在使用 Ant 时，Continuum 在项目的根目录查找项目的 build.xml 文件。如果使用不同的名称或者这个文件不在项目的根目录，可以修改这个设置。

虽然 Continuum 还是 CI 舞台上的新人，但是它以其易用性和对当前众多流行的版本控制系统和构建工具的支持，还是给这一领域带来了巨大的冲击。我希望在未来的版本中会有添加和查看报告的功能。

CruiseControl

CruiseControl 是 CI 服务器的老者。它已经用了有五年多了，在许多方面， CruiseControl 服务器 已经成为持续集成实践的同义词。出于完全坦白的目的，我应当提到，我也是 CruiseControl 的多年的老用户。

改进的安装

如果您从最后一次使用 CruiseControl 到现在已经有段时间，而且认为它的安装和配置是个负担，那么您可以看看最新版本。现有，有许多方式安装 CruiseControl。例如，如果使用 Windows，会发现最简单的方式是下载二进制可执行文件，然后运行它。不用担心，还可以下载源代码。

安装之后，CruiseControl 预先配置了一个配置文件，轮询 CVS 存储库并执行 Ant 构建脚本。同样也不需要安装 Web 服务器，因为 CruiseControl 也内嵌了 Jetty。

轮询版本控制系统

比起 Luntbuild 和 Continuum，CruiseControl 提供了对超过十种不同版本控制系统的支持。而且，CruiseControl 对这些工具中的许多定制命令也提供了支持。清单 1 是一个使用 CruiseControl config.xml 脚本轮询 Subversion 存储库的示例：

清单 **1.** 通过 **config.xml** 文件轮询存储库

```
<listeners>
  <currentbuildstatuslistener file="logs/${project.name}/status.txt"/>
</listeners>
<modificationset quietperiod="30">
  <svn RepositoryLocation="http://www.qualitylabs.org/svn/ambientorb/trunk"
    username="bfranklin"
    password="G0Fly@Kite"
  />
</modificationset>
```

执行构建脚本

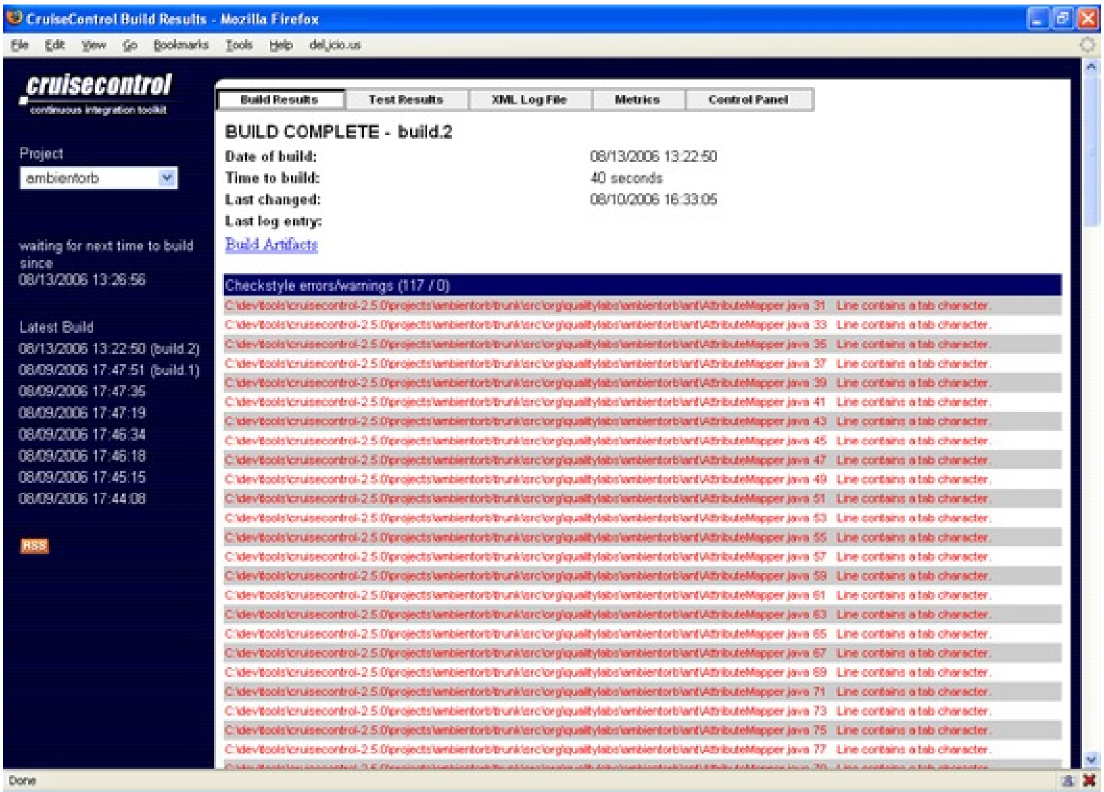
当在版本控制系统（例如 Subversion）中发现修改时，可以很容易地配置 CruiseControl 去执行委托的构建脚本。例如，清单 2 演示了从 config.xml 调用 Ant 脚本，它指示 CruiseControl 每 60 秒钟查询 Subversion 存储库一次，并执行另一个 Ant 脚本。 委托的构建脚本（没有显示）删除旧文件，从 Subversion 签出最新的源代码，并在代码上运行项目的构建脚本。

清单 **2.** 执行 **Ant** 构建脚本

```
<schedule interval="60">
  <ant anthome="apache-ant-1.6.5" buildfile="build-${project.name}.xml"/>
</schedule>
```

当设置了 CruiseControl 的这个方面并启动服务器之后，可以访问如图 3 所示的 CruiseControl Web 控制板：

图 **3. CruiseControl** 控制板



CruiseControl 控制板

要接收最新构建的反馈，可以把 `htmlemail` 插件添加到清单 3 所示的 config.xml 脚本。可以用 config.xml 文件配置更多反馈机制，例如发送文本消息、电子设备（通过 X10）、甚至即时消息。

清单 **3.** 用 **CruiseControl** 发送电子邮件

```
...
<plugin name="htmlemail"
  buildresultsurl="http://${env.COMPUTERNAME}/cruisecontrol/buildresults/${project.name}"
  mailhost="${smtp.server}"
  username="${mail.username}"
  password="${mail.password}"
  returnaddress="${buildmaster.email}"
  returnname="${buildmaster.name}"
  subjectprefix="${project.name} build"
  xsldir="webapps/cruisecontrol/xsl"
  css="${reportdir}/cruisecontrol.css"/>
   ...
  <htmlemail>
    <always address="${buildmaster.email}"/>
    <failure address="${buildmaster.email}"/>
  </htmlemail>
```

CruiseControl 提供了许多有用的特性，有强大的用户社区，极具扩展性。与本文中评估的其他工具相比，有些开发人员觉得 CruiseControl 不太容易使用。而另一方面，有些开发人员则发现用 XML 脚本进行修改提供了更好的控制。

Luntbuild

从面市年头上说，Luntbuild 位于 Continuum 和 CruiseControl 之间。比起 Continuum 和 CruiseControl，Luntbuild 的目标是为并行开发和用户管理之类的事情提供支持的构建管理服务器。它的整个配置是通过 Web 应用程序管理的，所以没有配置文件需要处理。它也有商业版可以使用，叫作 QuickBuild，商业版中包含用户支持。

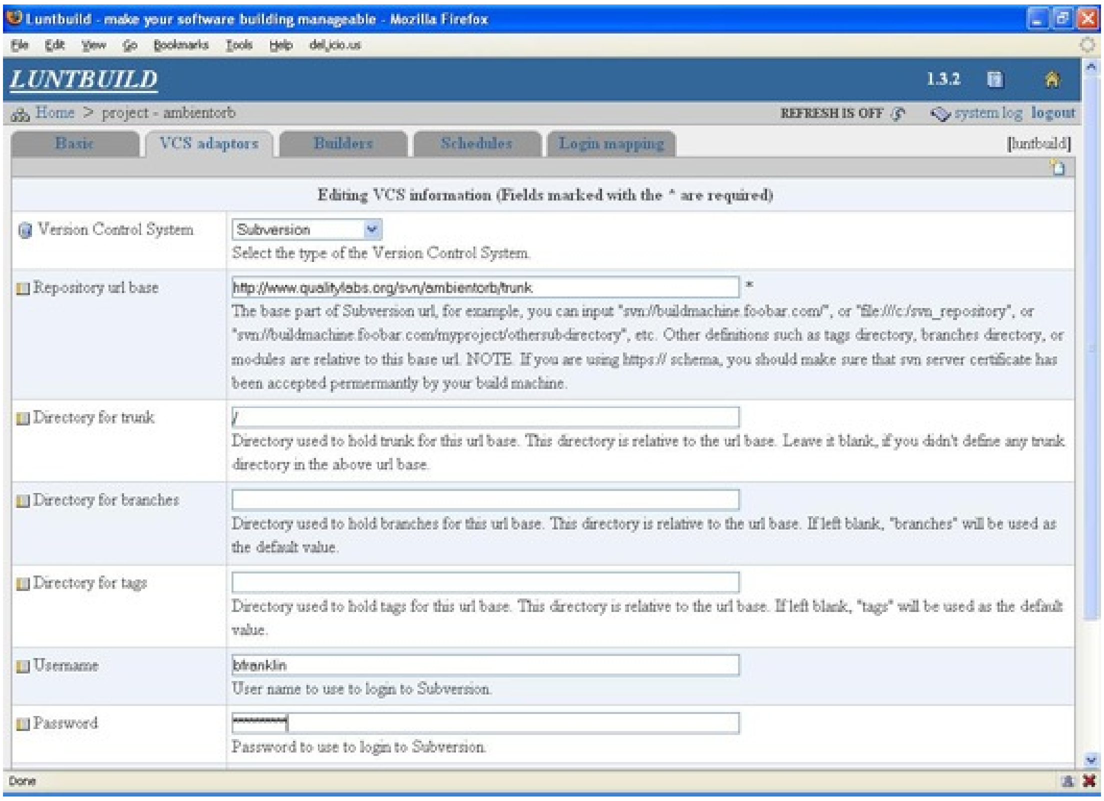
Jetty 不再必需

Luntbuild 提供了几种安装方式。您可能会发现最简单的方式是通过 GUI 安装。用 Web 应用程序配置和管理 Luntbuild；所以，需要确保正在运行一个能够处理 JSP 的 Web 服务器，像 Tomcat 或 Jetty。

版本控制轮询

Luntbuild 提供了对八种不同版本控制系统的支持，例如 CVS、Subversion、ClearCase 和 Perforce。图 4 演示了 Luntbuild 被设置成轮询 Subversion：

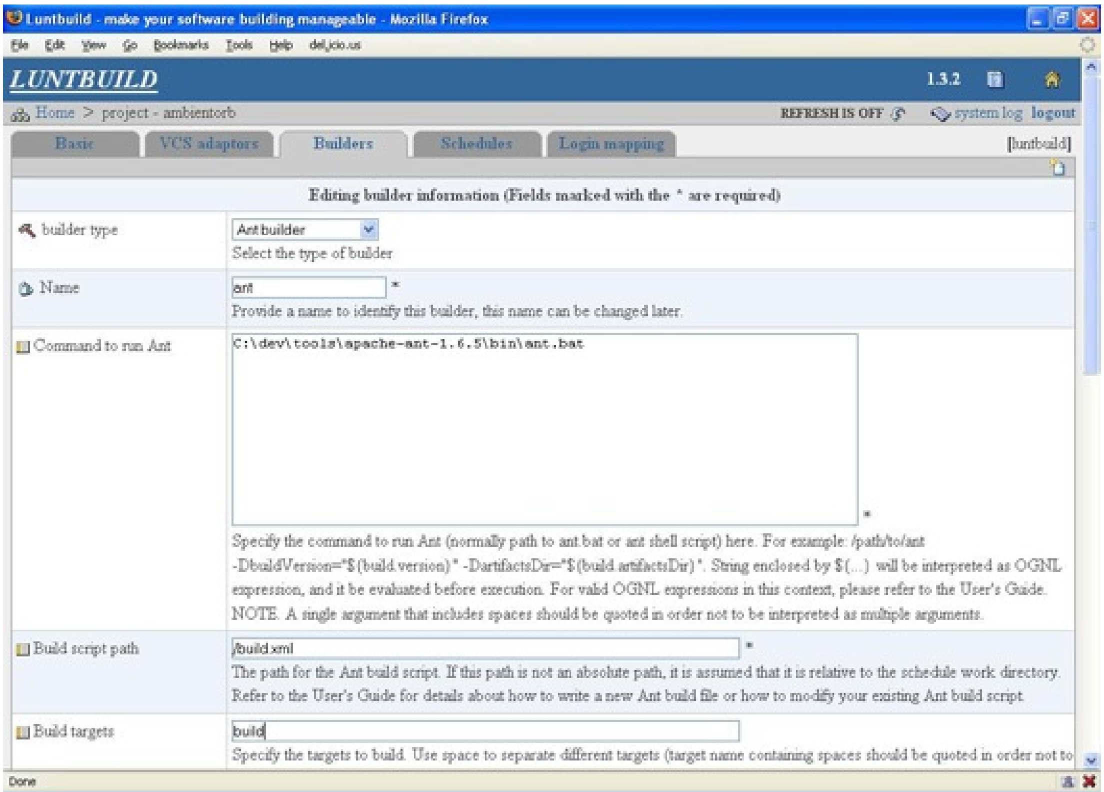
图 **4. Luntbuild** 轮询 **Subversion** 存储库



执行构建

Luntbuild 支持五种不同的构建平台，包括 Ant、Maven、Maven2、命令行和 rake （用来构建 Ruby 应用程序）。图 5 显示了 Ant 构建器的配置页面：

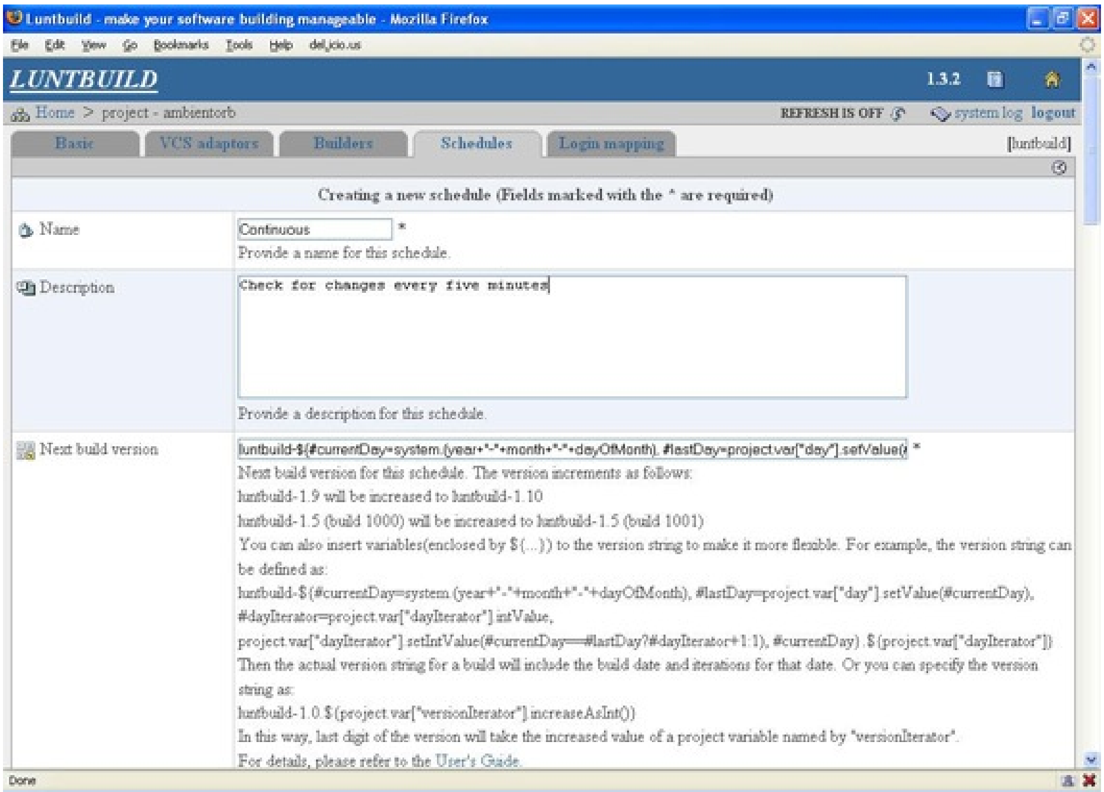
图 **5.** 用 **Luntbuild** 执行 **Ant** 脚本



构建安排

通过使用 Luntbuild 中的 Schedule 标签（如图 6 所示），可以设置 Luntbuild 多久轮询一次版本控制系统来获得修改。在这个标签上，还可以指定分配给每个构建的唯一构建标号。

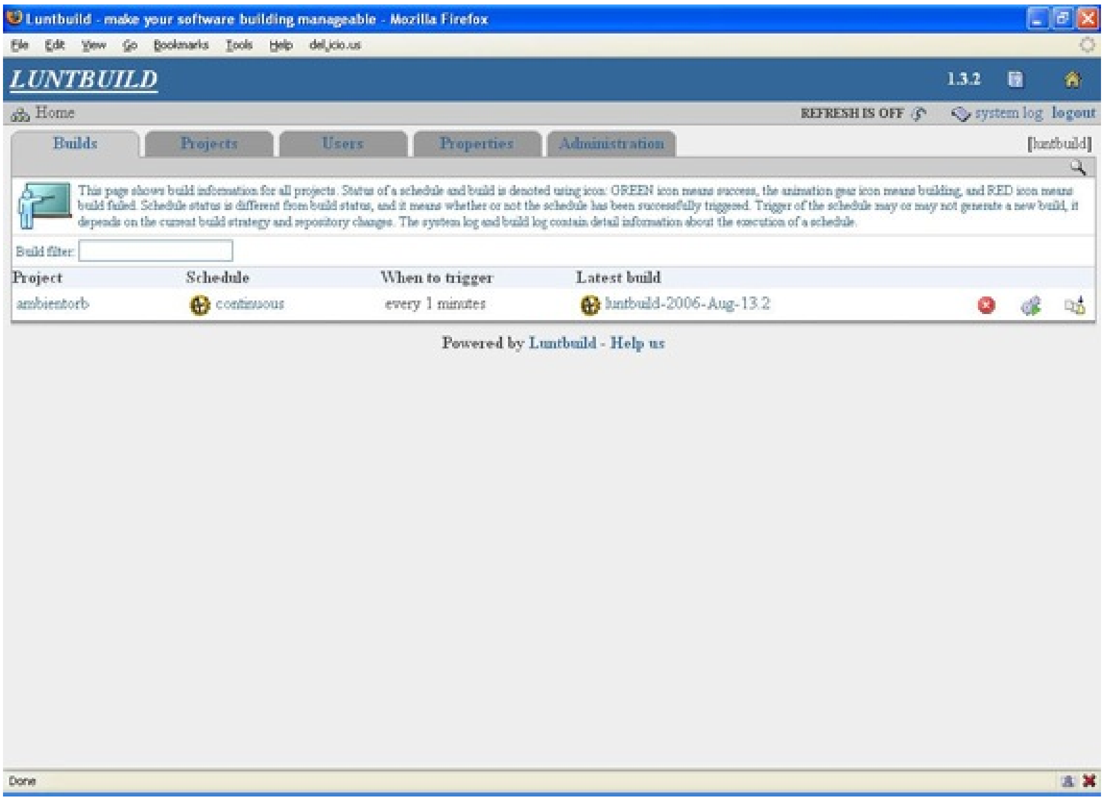
图 **6.** 在 **Luntbuild** 中安排构建



在 **Luntbuild** 中发布结果

配置了项目、版本控制系统适配器、构建和计划程序之后，就可以指定用户接收反馈的方式了。但是，Luntbuild 只内置了对电子邮件和即时消息的支持。另外，可以从 Luntbuild 的主页监视构建，如图 7 所示：

图 **7.** 从 **Luntbuild Web** 应用程序监视构建



Luntbuild 提供了一整套强大的功能，包括管理项目依赖项和大量的版本控制系统适配器。我认为工作流和用户界面可以简化，因为需要许多步骤来设置和配置工具。

CI 记分卡

在不理解具体需求的情况下，就推荐哪个工具合适是非常冒失的。每个服务器都有许多优秀的特性，而且就像我在开始时所提到的，仅仅因为某个 CI 服务器最适合某人，并不意味着它必然满足您的需求。

如果寻找的是易于使用的工具，请选择 Continuum。如果扩展性、灵活性和繁荣的用户社区对您很重要，请使用 CruiseControl。如果需要 Web 管理和扩展的用户支持选项，请考虑 Luntbuild。围绕这些服务器已经形成了开发“生态”系统，所以如果遗漏了某个特性，一般都会找到适合需求的扩展。

在表 2 中，是我根据自己的使用经验为所考察的每个 CI 服务器总结的特性、可靠性、寿命、目标环境和易用性这五个核心方面：

表 **2. CI** 服务器五个核心方面

	特性	可靠性	寿命	目标环境	易用性
Continuum	支持 Ant、Maven1 和 Maven2，以及 shell。				

使用 XML-RPC 和 SOAP 的远程管理能力；支持 Maven2；用户群；期待未来有附加的报告和反馈机制——不需要修改代码。 | 在 2005 年发布。期待通过它与 Apache 的关系，得到 Continuum 的更多消息。 | 通过 Apache Maven 的良好用户社区支持产品在市场上仍很新。 | Linux、Mac OS X、Solaris 和 Win32。 | 优秀的易用性和安装。 | | **CruiseControl** | 许多版本控制集成和扩展性。通过 JMX 控制的远程访问。多种反馈机制，包括 RSS、X10、Jabber 以及其他。 | 在 2001 年发布。在三个服务器中，CruiseControl 在开发中应用得最多。 | 繁荣的用户社区；每个迹象都表示 CruiseControl 还会存在一段时间。 | Windows 和 Unix；任何能运行 Java JVM 的平台。 | 易于安装。有些人宁愿不修改 XML 配置文件。 | | **Luntbuild** | 项目依赖项、标号、安全性组和并行开发。 | 在 2004 年发布。Luntbuild 提供扩展的用户支持选项。 | 用户社区不如 CruiseControl 活跃。 | 能够运行 JVM 和 servlet 容器的系统。 | 易于安装，但用户界面/工作流需要大大改进。基于 Web 的配置（不需要修改配置文件）。 |

我在本文中只评估了三个服务器；还有许多服务器可能更适合您的需求。但是既然您理解了如何挑选 CI 服务器，那么选择工作就应当很容易了。请继续关注下个月的文章，我将介绍在开发项目中经常会遇到的构建问题。

让开发自动化: 持续检查

把自己从软件检查员寻常的手工检查工作中解放出来

在开始新项目时，多数人计划在将代码投入生产发行之前审核它们；但是，当提交日程超越了其他因素时，审核常常成为第一个被抛弃的实践。如果能够自动执行其中一些审核，那么情况又会怎样呢？在新系列 "让开发自动化" 的第一篇文章中，开发自动化专家 Paul Duvall 首先将研究如何自动化检查器（例如 CheckStyle、JavaNCSS 和 CPD）、如何改进开发过程，以及应该什么时候使用它们。

代码检查可以采用不同的形式。有些企业使用正式的同级评审（peer review），在该评审过程中，开发人员要为代码提供同级评价，并提供改进意见；其他一些企业使用结对编程；还有一些人则考虑使用高级设计决策和推荐的代码改进。有些团队会在将代码提交到版本控制存储库之前，让其他开发人员用 "桌面检查" 的形式审查他们的代码。

不论企业采用哪种方式进行代码检查，有一件事是肯定的：它们几乎都是手工过程。正如我多年所观察到的，手工过程很容易出错，如果工作紧张，就会忘记自己正在做什么。但是，软件检查不必总是手工完成；实际上，有一大堆开源工具和商业工具（我称之为软件检查器），可以用它们很方便地对代码进行静态分析（这些工具也称为静态分析工具）。

使用软件检查器，代码检查可以通过 Ant 或 Maven 这样的构建工具来自动完成。通过使用这种自动化，一些低级的源代码细节（如编码标准、复杂性和重复程度等）的处理成为了机器的职责。这种职责转移通过将重点转向更高级的开发方面（比如设计和长期维护问题）提高了人们手工检查的效率。

关于本系列

作为开发人员，我们的工作就是为用户提供自动化处理；但是，我们中的许多人却忽视了自动化自己的开发过程的机会。出于这个目的，让开发自动化 这一系列的文章专门研究了自动化软件开发过程的实际应用，并教您什么时候 和如何 成功地应用自动化。

用软件检查器来解决问题

软件检查器有很多，范围从复杂的商业工具到简单的开源替代工具。但是，我发现其中三个特别有用，它们是 CheckStyle、CPD 和 JavaNCSS（请参阅 参考资料）：

- CheckStyle 报告与项目预定的编码标准的偏离度。
- CPD 报告代码重复。
- JavaNCSS 可以帮助团队专注于更高级的代码复杂性领域。

所有这些工具都可以免费得到，而且易于集成到 Ant 或 Maven 这样的构建平台中，因此它们极易集成到持续集成（CI）系统中。

什么是持续集成？

持续集成（CI）是一种实践，可以让团队在持续的基础 上收到反馈并进行改进，不必等到开发周期后期才寻找和修复缺陷。诸如 CruiseControl 之类的检查工具是在后台运行的，它们轮询版本控制存储库，从中寻找更改之处。当发现某一更改时，这类工具就会通过 Ant 执行预定义的构建脚本。持续检查借助持续集成的实践得以改进。

检查器样式标准

我参与了几家公司的委员会，花了很多时间来定义编码标准。有一次，我们定义了几乎没人遵守的 25 页的编码标准文档。结果，我们的代码审查非常痛苦，因为团队要花时间来挑错，看看开发人员是否使用了两个空格规则（与四个空格规则相对），以及是否将参数声明为 `final` 。如果把这些低级检查工作交给软件检查器，我们会节省许多宝贵的时间。

使用自动的源代码检查工具（如 CheckStyle）可以提供执行可配置规则集的简单途径，规则集可以根据项目的编码标准而定。此外，可以通过 CheckStyle 插件在 IDE 中运行 CheckStyle，CheckStyle 通过它的 Ant 任务或 Maven 插件直接集成到构建脚本中。CheckStyle 发现的所有规则偏离都会以报告的形式显示，清楚地指出哪一个文件违规。除此之外，还可以将 Ant 和 Maven 配置成与 CheckStyle 相呼应，这样，在违反规则时，构建工作就会失败。

例如，清单 1 演示了 Ant 构建脚本的一个代码段，它运行 CheckStyle 生成 HTML 报告。请注意 `failOnViolation` 属性，在这里它被设为 `false` 。如果将该属性设为 `true` ，则在发现任何 源代码违规时，构建都会失败。

清单 1. 将 CheckStyle 用于 Ant

```
<taskdef resource="checkstyletask.properties" classpath="${checkstyle.jar}"/>
<checkstyle config="${basedir}/cs-rules.xml" failOnViolation="false">
  <formatter toFile="${checkstyle.data.file}" type="xml" />
  <fileset casesensitive="yes" dir="${src.dir}" includes="**/*.java" />
</checkstyle>
<xslt taskname="checkstyle"
  in="${checkstyle.data.file}"
  out="${checkstyle.report.file}"
  style="${checkstyle.xsl.file}" />
```

在清单 1 中， `config` 被设置成 `cs-rules.xml` ，这表示将根据源代码目录（在 `fileset dir` 属性中）以递归的方式运行规则文件。 `xslt` 任务接受根据 `formatter toFile` 属性生成的文件。该任务使用 XSL 文件 checkstyle.xsl 将清单 1 生成的 XML 转换成一个可读的 HTML 文件，XSL 文件 checkstyle.xsl 包含在 CheckStyle 安装文件中。

清单 2 是 CheckStyle 规则文件的示例片段。CheckStyle 可以在代码基上运行 120 多个规则。

清单 **2.** 在 **cs-rules.xml** 文件中定义的 **CheckStyle** 规则

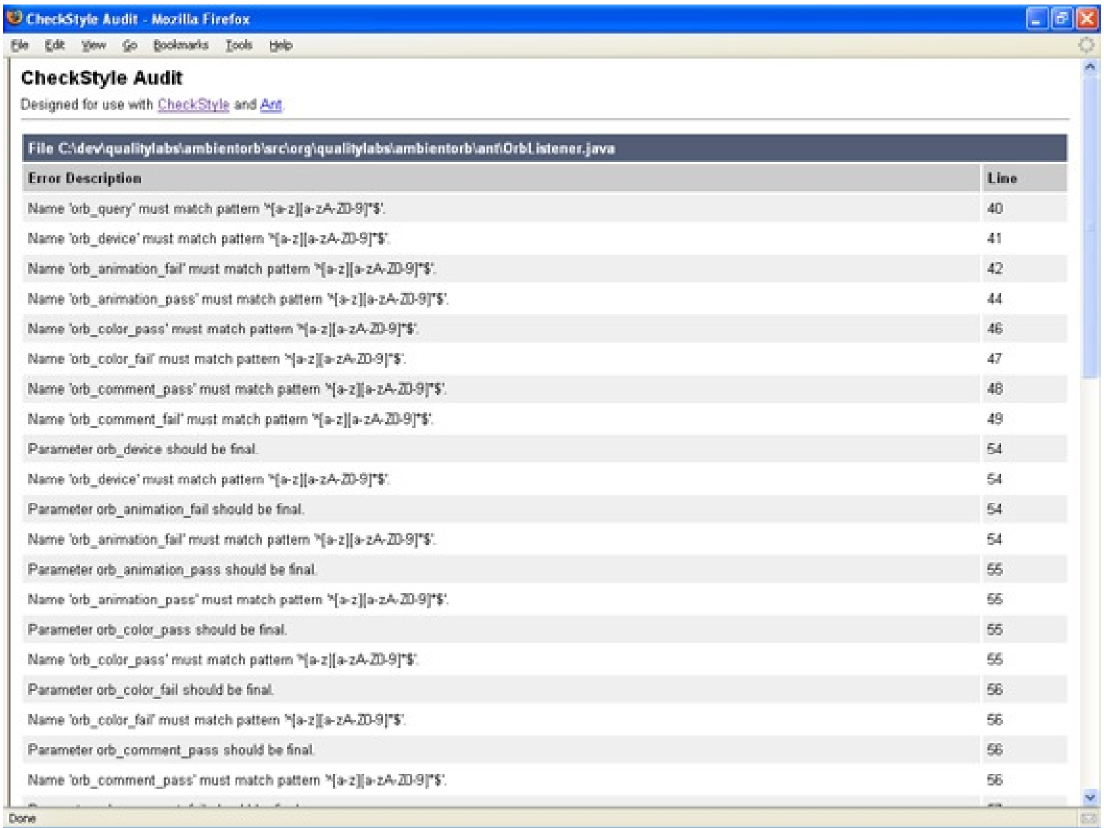
```
<module name="Checker">
  <module name="TreeWalker">
    <property name="cacheFile" value="target/checkstyle.cache"/>
    <property name="tabWidth" value="4"/>
    <module name="ImportOrder">
      <property name="ordered" value="true"/>
      <property name="separated" value="true"/>
    </module>
    <module name="LineLength">
      <property name="max" value="120"/>
    </module>
    <module name="FileLength">
      <property name="max" value="400"/>
    </module>
    <module name="UnusedImports"/>
  </module>
</module>
```

每个 CheckStyle 规则都是一个模块。例如，`LineLength` 模块建立的规则是：所有行中的字符都不得超过 120 个字符，否则 CheckStyle 就生成错误。

可以用定制规则扩展 CheckStyle，所以也可以容易地实施编码规则。而且，它是自动执行的！

图 1 显示了使用 Ant 生成的 CheckStyle 报告的一个示例，这是 清单 1 中所演示的 `xslt` 任务：

图 **1. CheckStyle HTML** 报告



CheckStyle Audit - Mozilla Firefox

CheckStyle Audit

Designed for use with CheckStyle and Ant.

File C:\dev\qualitylabs\ambientorb\src\org\qualitylabs\ambientorb\ant\OrbListener.java

Error Description	Line
Name 'orb_query' must match pattern '^[a-z][a-zA-Z0-9]*$'.	40
Name 'orb_device' must match pattern '^[a-z][a-zA-Z0-9]*$'.	41
Name 'orb_animation_fail' must match pattern '^[a-z][a-zA-Z0-9]*$'.	42
Name 'orb_animation_pass' must match pattern '^[a-z][a-zA-Z0-9]*$'.	44
Name 'orb_color_pass' must match pattern '^[a-z][a-zA-Z0-9]*$'.	46
Name 'orb_color_fail' must match pattern '^[a-z][a-zA-Z0-9]*$'.	47
Name 'orb_comment_pass' must match pattern '^[a-z][a-zA-Z0-9]*$'.	48
Name 'orb_comment_fail' must match pattern '^[a-z][a-zA-Z0-9]*$'.	49
Parameter orb_device should be final.	54
Name 'orb_device' must match pattern '^[a-z][a-zA-Z0-9]*$'.	54
Parameter orb_animation_fail should be final.	54
Name 'orb_animation_fail' must match pattern '^[a-z][a-zA-Z0-9]*$'.	54
Parameter orb_animation_pass should be final.	55
Name 'orb_animation_pass' must match pattern '^[a-z][a-zA-Z0-9]*$'.	55
Parameter orb_color_pass should be final.	55
Name 'orb_color_pass' must match pattern '^[a-z][a-zA-Z0-9]*$'.	55
Parameter orb_color_fail should be final.	56
Name 'orb_color_fail' must match pattern '^[a-z][a-zA-Z0-9]*$'.	56
Parameter orb_comment_pass should be final.	56
Name 'orb_comment_pass' must match pattern '^[a-z][a-zA-Z0-9]*$'.	56

Done

用 IDE 怎么样？

我听到的一个普遍反应是这些检查可以在 IDE 中运行。尽管确实如此，并且极力推荐这样做，但是拥有一台对源代码运行一组公用规则集（使用 Ant 这样的工具）的单独机器的好处是：一致性更好。

用 CPD 进行重复性检查

面向对象编程的一个主要目的是通过创建可以适应不同上下文的可重用对象来促进重用。但是，通常使用的是“复制-粘贴”编程，并将它伪装成“重用”。有时，出现这种情况是因为开发人员不知道有可重用的组件，而其他时候，则是因为提供了“快速修复”。所以，如果需要进行修改，则必须将更改应用到所有 副本和变体中（当然，如果没有工具很难完成）。这种手工方式带来了维护困难并很容易导致遗漏，甚至会带来缺陷。

CPD 是流行的开源静态分析工具 PMD 的一部分，它报告代码基中重复行的数量。此外，CPD 的标志阈值是可配置的，这意味着可以修改 CPD 建议的重复行数。例如，如果将阈值设置成 100 个标志，CPD 就会在至少 重复了 100 个标志的时候显示一个实例。不过请记住，源代码总是会包含一些重复部分；CPD 只是为团队提供了一种调查值得考虑的代码重复领域的手段，在这个领域内，可以采取纠正措施，例如有目的的重构。

清单 3 演示将 CPD 用于 Ant 的实际使用。CPD 任务需要的第一个选项是 `minimumTokenCount` 属性，该属性用于指定要比较的标志的数量（最小值是一个标志）。另外，可以设置 CPD 忽略某些代码选项，例如标识符和标量。在这个示例中，Ant `fileset` 类

型指定了 CPD 文件应该分析哪些文件，以及应该忽略哪些文件。

清单 **3.** 将 **CPD** 用于 **Ant**

```
<taskdef name="cpd"
  classname="net.sourceforge.pmd.cpd.CPDTask" classpathref="pmd.classpath" />
  <cpd minimumTokenCount="100"
        format="xml"
        language="java"
        ignoreIdentifiers="true"
        ignoreLiterals="true"
        outputFile="${basedir}/cpd.xml">
        <fileset dir="${src.dir}">
          <include name="**/*.java" />
          <include name="**/*.jsp" />
          <include name="**/*.xml" />
          <exclude name="**/*Test*.java" />
        </fileset>
  </cpd>
```

如图 2 所示，cpd.xml 是用清单 3 中的 `outputFile` 属性产生的输出。以后我还将向 CPD 添加 XSL，以生成 HTML 报告。

图 **2.** 显示代码重复违规的 **CPD XML** 输出

除了最明显的复制-粘贴违规之外，在所有地方手工查找重复代码也非常具有挑战性。但是，使用诸如 CPD 之类的工具，就可以迅速找出重复代码，然后对此进行改进。

路径是什么？

源代码路径 也叫做流程控制语句，它指示在方法中更改流程的代码。这类示例包括条件语句（如 `if` 语句），或者循环构造（如 `while` 和 `for` ）。

用 **JavaNCSS** 检查复杂性

许多研究已经证实，随着方法中路径数量的增加，理解和维护同一方法的难度也随之增加。结果，代码越难以理解和维护，就越容易出现缺陷。

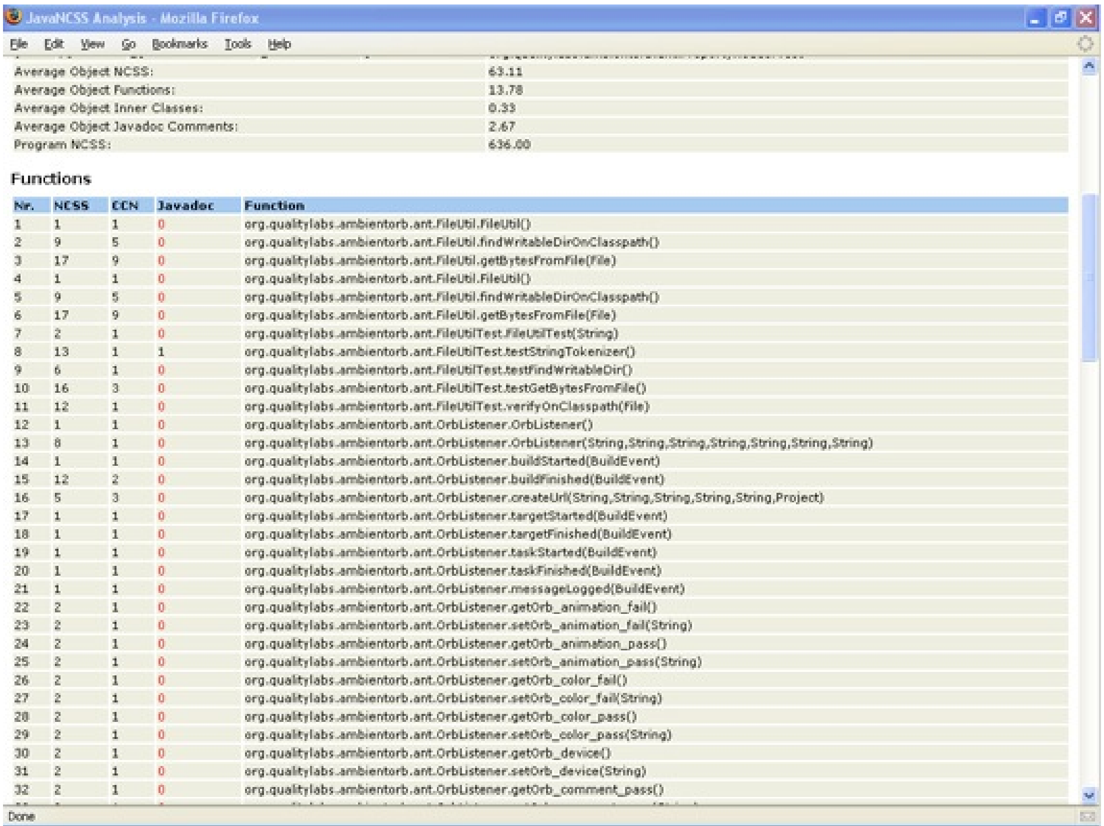
JavaNCSS 是一个免费工具，提供了不同的代码测量，例如非注释性源代码语句（或代码行）的数量、所有分析过的方法的圈复杂度量值（cyclomatic complexity number）。清单 4 演示了 JavaNCSS 的使用，可以看到，它不是很难使用。只要将 JavaNCSS 指向一个包含源代码的目录，让它自己处理就行了！

清单 **4.** 将 **JavaNCSS** 用于 **Ant**

```
<path id="javancss.classpath">
  <fileset dir="${lib.dir}" />
</path>
<taskdef name="javancss" classpathref="javancss.classpath"
  classname="javancss.JavancssAntTask">
</taskdef>
<javancss srcdir="${src.dir}"
      generateReport="true"
      outputfile="${javancss.report.dir}/javancss_metrics.xml"
      format="xml"/>
```

像多数软件检查器一样，这里生成了一个 XML 文件，可以通过 XSLT 很容易地将它转换成一个 HTML 报告，就像图 3 中所做的那样：

图 **3. JavaNCSS HTML** 报告


JavaNCSS Analysis - Mozilla Firefox

Average Object NCSS:	63.11
Average Object Functions:	13.78
Average Object Inner Classes:	0.33
Average Object Javadoc Comments:	2.67
Program NCSS:	636.00

Functions

Nr.	NCSS	CCN	Javadoc	Function
1	1	1	0	org.qualitylabs.ambientorb.ant.FileUtil.FileUtil()
2	9	5	0	org.qualitylabs.ambientorb.ant.FileUtil.findWritableDirOnClasspath()
3	17	9	0	org.qualitylabs.ambientorb.ant.FileUtil.getBytesFromFile(File)
4	1	1	0	org.qualitylabs.ambientorb.ant.FileUtil.FileUtil()
5	9	5	0	org.qualitylabs.ambientorb.ant.FileUtil.findWritableDirOnClasspath()
6	17	9	0	org.qualitylabs.ambientorb.ant.FileUtil.getBytesFromFile(File)
7	2	1	0	org.qualitylabs.ambientorb.ant.FileUtilTest.FileUtilTest(String)
8	13	1	1	org.qualitylabs.ambientorb.ant.FileUtilTest.testStringTokenizer()
9	6	1	0	org.qualitylabs.ambientorb.ant.FileUtilTest.testFindWritableDir()
10	16	3	0	org.qualitylabs.ambientorb.ant.FileUtilTest.testGetBytesFromFile()
11	12	1	0	org.qualitylabs.ambientorb.ant.FileUtilTest.verifyOnClasspath(File)
12	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.OrbListener()
13	8	1	0	org.qualitylabs.ambientorb.ant.OrbListener.OrbListener(String,String,String,String,String,String,String)
14	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.buildStarted(BuildEvent)
15	12	2	0	org.qualitylabs.ambientorb.ant.OrbListener.buildFinished(BuildEvent)
16	5	3	0	org.qualitylabs.ambientorb.ant.OrbListener.createUrl(String,String,String,String,String,Project)
17	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.targetStarted(BuildEvent)
18	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.targetFinished(BuildEvent)
19	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.taskStarted(BuildEvent)
20	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.taskFinished(BuildEvent)
21	1	1	0	org.qualitylabs.ambientorb.ant.OrbListener.messageLogged(BuildEvent)
22	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.getOrb_animation_fail()
23	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.setOrb_animation_fail(String)
24	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.getOrb_animation_pass()
25	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.setOrb_animation_pass(String)
26	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.getOrb_color_fail()
27	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.setOrb_color_fail(String)
28	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.getOrb_color_pass()
29	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.setOrb_color_pass(String)
30	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.getOrb_device()
31	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.setOrb_device(String)
32	2	1	0	org.qualitylabs.ambientorb.ant.OrbListener.getOrb_comment_pass()

理解应用程序代码复杂性可没有那么容易。实际上，在某些情况下，复杂性的值可能会令人误解。例如，有些工具为 `switch` 语句提供较高的值。但是我发现，使用 JavaNCSS 这样的工具，有助于降低高复杂性的领域，最终提高代码的可理解性和可维护性。

对每个更改都进行检查，将缺陷控制在限度内

每当团队成员向版本控制存储库提交更改时，代码就会发生改变。但它是怎么改变的呢？被修改的代码是受到复制-粘贴工作的影响吗？复杂性会增加吗？了解这些的惟一方式就是在每次签入 时都运行软件检查器。此外，在持续的基础上接受目前为止讨论过的每种风险的反

馈，这是一种自动 让代码基进行健康检查的一种可靠方式！

Cruise 与 CI

CruiseControl 是 Java™ 社区使用最广的开源 CI 工具之一。这个工具被配置成在后台运行，用于检索版本控制存储库，例如 CVS。在发现源代码更改时（例如，有人签入了代码），CruiseControl 就会执行源代码签出，并运行预定义的构建脚本。

所以，只要 版本控制存储库中发生了变化，团队就可以运行 CheckStyle、CPD 和 JavaNCSS 这样的软件检查器。这种能力允许团队长时间地进行监视和执行检查，通过使用许多 CI 工具，团队可以使用报告、电子邮件甚至使用 Ambient Orb 这样的设备生成报告（参见 图 5）。

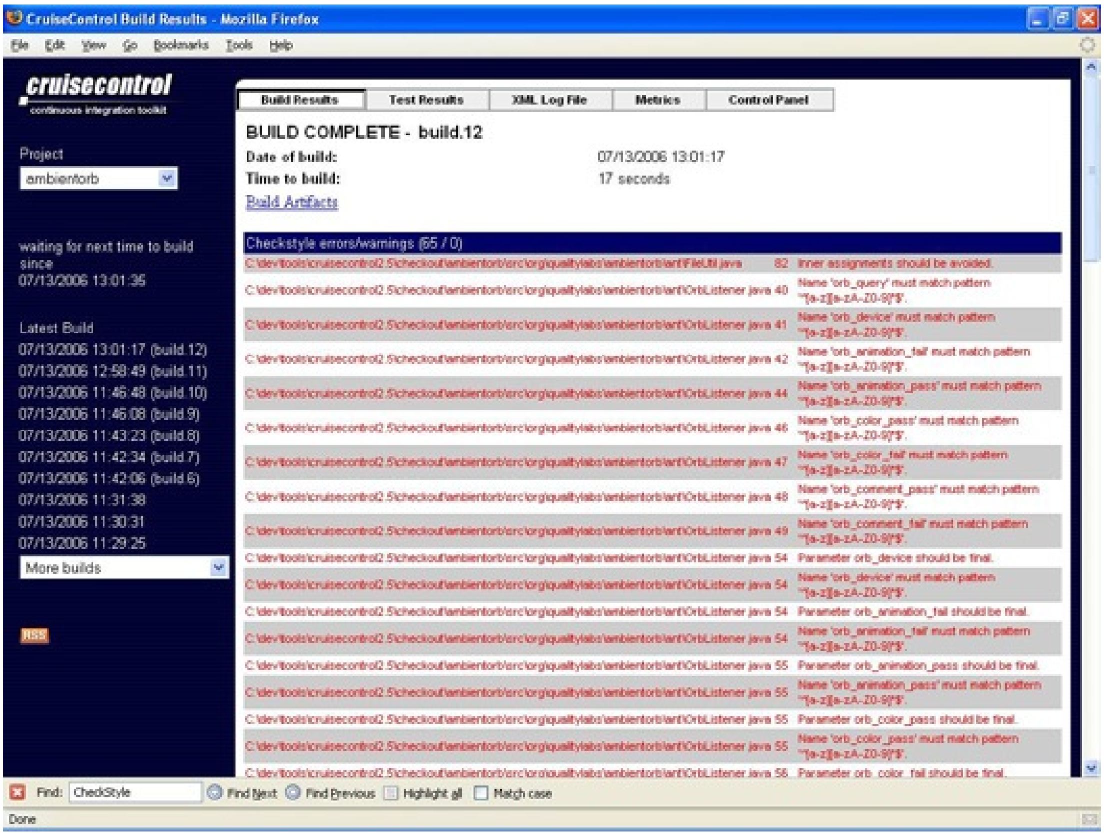
清单 5 演示了使用 CruiseControl 的配置文件（通常名为 config.xml）在 CruiseControl 控制板上显示 CheckStyle 报告的结果。其他软件检查器使用类似的语法将结果合并到 CruiseControl 仪表板中。

清单 **5.** 在 **config.xml** 中用 **CruiseControl** 记录 **CheckStyle**

```
<listeners>
  <currentbuildstatuslistener file="logs/${project.name}/status.txt"/>
</listeners>
<modificationset quietperiod="30">
  <svn RepositoryLocation="http://www.qualitylabs.org/svn/ambientorb"
    username="[username]"
    password="[password]"
  />
</modificationset>
<schedule interval="60">
  <ant anthome="apache-ant-1.6.5" buildfile="build-${project.name}.xml"/>
</schedule>
<log>
  <merge dir="merge dir="checkout/${project.name}/_reports/checkstyle" />
</log>
```

图 4 显示了用 CruiseControl 和 CheckStyle 生成的示例报告。请注意，可以配置 CruiseControl，显示其他工具（像 CPD 和 JavaNCSS）的报告，而且通过对每个源代码更改 都运行这些报告，团队可以实时地积极改进代码，不必等到周期末期。

图 **4.** 与 **CruiseControl** 集成的 **CheckStyle** 报告



Ambient 反馈

对于使用 CI 工具持续运行软件检查器而言，最酷的事就是团队有了无数任意使用的通知机制。有时，构建可能并没有失败，但是有些事 的变化要求早些而不是晚些采取纠正行动。例如，实际上可以很容易地配置一个设备（就像 Ambient Orb），在代码复杂度有所上升时，或者在违反一定数量的代码标准时，使用该设备改变颜色。

清单 6 使用了 Ambient Orb Ant 任务和 Ruby 脚本，在 20 个以上的类超过 300 个源代码行（SLOC）时，就改变 Orb 的颜色和动画。在这个示例中，我选择在类满足条件时将 orb 的颜色改成 `magenta`，将动画改成 `crescendo`。

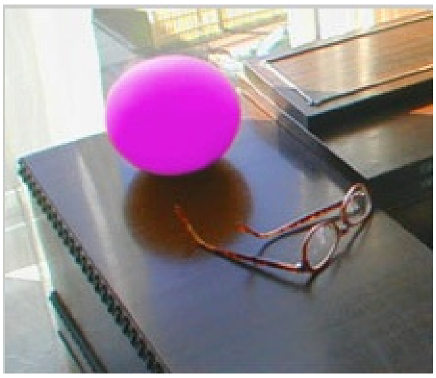
清单 6. 委托 Ant 构建文件处理 CruiseControl 和 Ambient Orb

```
<target name="checkSloc" >
  <exec dir="${basedir}" executable="cmd.exe">
    <arg line="/c ${config.dir}/javancss/SlocThreshold.rb
        ${reports.javancss.dir}/javancss_metrics.xml 20 ${javancss.file}"/>
  </exec>
  <available file="${basedir}/${javancss.file}" property="sloc.exceeded"/>
  <antcall target="notifyOrb" />
</target>

<target name="notifyOrb" if="sloc.exceeded">
  <taskdef classname="org.qualitylabs.ambientorb.ant.OrbTask"
   name="orb" classpathref="orb.class.path"/>
  <orb query="http://myambient.com:8080/java/my_devices/submitdata.jsp"
    deviceId="AAA-9A9-AA9"
    colorPass="green"
    colorFail="magenta"
    animationPass="none"
    animationFail="crescendo"
    commentFail="SLOC+Exceeded" />
</target>
```

图 5 演示了代码基中有太多较大的类时，orb 看起来的样子：

图 **5. Ambient Orb**



使用这种 Ambient Orb 通知方法，就不会收到无穷尽的电子邮件，只要看一眼 *SLOC Threshold Orb*，就可以快速了解我有多少个大型类，它们提醒我要重定向重构工作。当然，组合 ambient orb 与软件监视器（稍带些创意）也会创造无穷的可能性！

在合适的时候自动化

请记住，可以选择许多不同的软件检查器。虽然这里分析了三个我喜欢的软件检查器，但我还是被 JDepend、PMD 和 FindBugs 工具（后两个检查器在 developerWorks 上已详细讨论过）所打动。一旦开始持续检查过程，插入一个新的检查器只是几分钟的事而已。

持续检查不会也不应该消除手工软件检查；但是，通过对版本控制存储库中的每个更改都运行一套自动检查工具，将要执行的 这些手工检查会带来更高的生产率和效率。而且，持续检查带来的额外好处有：风险在每一步上都可以降低，而不必等到项目后期。

下一个月，我将深入研究一些更有趣的持续集成工具，例如 CruiseControl、Luntbuild 和 Continuum，并让您确定哪一个工具最适合您。自动化从未 这么有趣过！

下载

描述	名字	大小
Software Inspection examples in Ant	j-ap08016.zip	2MB